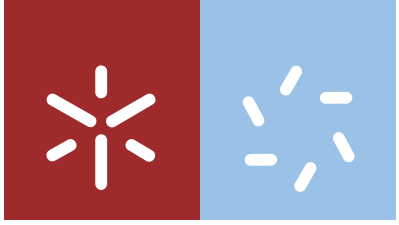
Universidade do Minho
Escola de Ciências

Eliana Tiba Gomes Grande

E-learning: Geração automática de exercícios para sistemas de ensino personalizado de Programação

outubro de 2017

UMinho | 2017 Eliana Tiba Gomes Grande **E-learning: Geração automática de exercícios para sistemas de ensino personalizado de Programação**



Universidade do Minho
Escola de Ciências

Eliana Tiba Gomes Grande

E-learning: Geração automática de exercícios para sistemas de ensino personalizado de Programação

Tese de Douturamento em Ciências
Especialização em Matemática

Trabalho efetuado sob a orientação do
Professor Doutor Gueorgui Vitalievitch Smirnov
e do
Professor Doutor José João Antunes Guimarães Dias de Almeida

outubro de 2017

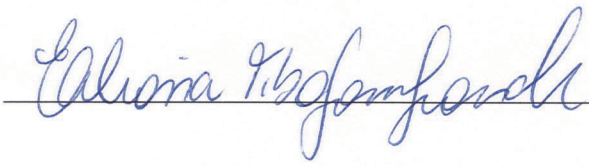
DECLARAÇÃO DE INTEGRIDADE

Declaro ter atuado com integridade na elaboração da presente tese. Confirmo que em todo o trabalho conducente à sua elaboração não recorri à prática de plágio ou a qualquer forma de falsificação de resultados.

Mais declaro que tomei conhecimento integral do Código de Conduta Ética da Universidade do Minho.

Universidade do Minho, 27 de Outubro de 2017

Nome completo: Eliana Tiba Gomes Grande

Assinatura: 

Dedicatoria

Dedico ao Rei dos séculos, ao único Deus sábio, seja toda honra e toda a glória para todo o sempre e sempre. Amém.

Agradecimentos

Primeiramente, agradeço a Deus Pai por nos abençoar nesta grande jornada em busca do conhecimento.

Agradeço à minha família que não mediram esforços para nos apoiar em todos os momentos, em especial meu marido Maxwell, meus filhos Gabriel e Geovana, e meus pais, José Reinaldo e Nobuko.

Agradeço igualmente os meus professores orientadores o Dr. Gueorgui V. Smirnov e o Dr. José João A. G. D. de Almeida pela paciência e dedicação.

Agradeço de todo o meu coração.

Resumo

Esta tese tem como objetivo principal o desenvolvimento de bases teóricas para a elaboração de sistemas de *e-learning* centrados no domínio de Programação, nomeadamente em gramáticas e linguagens independentes de contexto. A aprendizagem dos formalismos para descrição de linguagens (de programação) levanta dificuldades ligadas ao facto de envolver mecanismos de abstração e lidar com objetos complexos. No que toca a gramáticas e formalismos linguísticos, constatamos que as técnicas habitualmente usadas em e-learning, esbarram com diversos problemas de geração e de capacidade de verificação automática de exercícios.

Nesta dissertação desenvolve-se e discute-se formalismo de metagramáticas para geração de exercícios, teoremas de distinguibilidade para gramáticas independentes de contexto, algoritmos de comparação de gramáticas independentes de contexto, análise experimental da aplicabilidade de algoritmos de comparação de gramáticas independentes de contexto, um conjunto de ferramentas de processamento de gramáticas.

Os resultados obtidos mostram que o domínio das gramáticas independentes de contexto pode ser incluído em sistemas de e-learning, permitindo geração de exercícios e a sua avaliação automáticas.

Abstract

This thesis has as a main objective the development of theoretical bases for the creation of *e-learning* systems concerning the domain of Programming, namely context-free grammars and languages. The study of the (programming) languages description formalism rises difficulties connected with the fact of involvement of abstraction mechanisms and dealing with complex objects. In what concerns grammars and linguistic formalisms, we observed that the techniques commonly used in e-learning face several problems of generation and automatic assessment of exercises.

In this dissertation we develop and discuss metagrammar formalism for exercises generation, distiguishability theorems for context-free grammars, algorithms for comparison of context-free grammars, experimental analysis of the applicability of algorithms for comparison of context-free grammars, a set of tools for grammar processing.

The obtained results show that the domain of context-free grammars can be included in e-learning systems, allowing generation and automatic assessment of exercises.

Conteúdo

1 Introdução **1**

1.1 Geração automática de exercícios 2
1.1.1 Estado da arte 2
1.1.2 Objetivos 3
1.2 Geração de exercícios – metagramáticas 4
1.3 Verificação automática 5
1.4 Trabalho experimental 8
1.5 Resultados principais da tese 8
1.6 Organização do trabalho 9

2 Conceitos básicos **11**

2.1 Linguagens e gramáticas 11
2.1.1 Séries formais de potências 11
2.1.2 Sistemas de equações correspondentes a uma gramática 12
2.1.3 Forma normal de Chomsky 12
2.1.4 Forma normal de Greibach 12
2.2 Aplicações lineares 13
2.3 Teorema do ponto fixo 13

3 Geração de exercícios de gramáticas **15**

3.1 Meta-gramatica 17
3.1.1 Estrutura de uma meta-gramática 17
3.1.2 Exemplo 18
3.1.3 Atributos da meta-gramática 21
3.1.4 Exemplo com atributos complexos 22
3.2 Módulo **Mgbeg** 24
3.3 Conclusão 25

4 Comparação de gramáticas **27**
4.1 Teoremas de distinguibilidade 29
4.1.1 Teorema geral de distinguibilidade 30
4.1.2 Exemplos 33
4.1.3 Testes para análise de linguagens finitas sobre alfabeto $\{a, b, c\}$ 33
4.1.4 Teoremas de distinguibilidade baseados na comparação de palavras curtas 39
4.2 Sistema de equações não-lineares matriciais 39
4.2.1 Convergência do método iterativo 40
4.2.2 Outros casos de aplicação do método iterativo 42
4.3 Limitações do método iterativo 45
4.4 Conclusão 48

5 Sistema / ferramentas **49**
5.1 Definição de gramáticas 49
5.2 Reconhecimento de frases 53
5.3 Geração de frases 56
5.4 Comparação de gramáticas 62
5.4.1 Função **TMatrixGen** 63
5.4.2 Função **solverGram** 63
5.4.3 Função **matrixCompare** 64
5.4.4 Função **gramequiv** 66
5.5 *Pretty print* de gramáticas 67

6 Conclusão **71**
6.1 Conclusão e trabalhos futuros 71

Capítulo 1

Introdução

O ensino moderno mais e mais frequentemente vem recorrendo a uma progressiva utilização das mais diversas ferramentas ligadas a Informática. Isto leva ao desenvolvimento de mecanismos que motivem o aluno, estimulem a sua curiosidade e autonomia e também potencializem o sucesso escolar. As relações entre a Tecnologia da Informação e o ensino autoregulado têm sido, em diversos campos, alvo de estudo e há relatos de uma grande variedade de casos bem sucedidos [18].

A geração e apresentação de problemas está omnipresente na educação ou processos de ensino, quer numa abordagem objetivista quer construtivista, quer se recorra a treino, quer a tutoria, quer se use a Internet quer se use papel e lápis, havendo em todas as situações a necessidade de um método para selecionar e gerar exercícios e problemas para o aluno ([13], p. 237).

O Ensino Assistido por Computador (EAC), surgiu nos finais da década de 60, como uma sequência de problemas apresentados aos alunos, receção de respostas e avaliação do desempenho daqueles. Contudo, depressa se percebeu a necessidade de criar ferramentas mais poderosas para gerar e adaptar exercícios, surgindo como natural o uso de técnicas de inteligência artificial ([13], p. 238).

Os trabalhos prévios de Koffman na geração de problemas (citados em [13], p. 238) mostram algumas características normalmente encontradas em sistemas posteriores, tais como a seleção de um domínio formal, o uso de gramáticas formais ou a necessidade de mecanismos adicionais com o objetivo de controlar as frases geradas pela gramática. Este início promissor não foi seguido por uma enchente de sistemas geradores em EAC ou Sistemas Tutoriais Inteligentes (STI), visto que, segundo os autores, com a tecnologia da época o uso de tais programas não era apelativo nem economicamente viável.

Com a evolução tecnológica e desenvolvimento da Informática, assistiu-

se a uma mudança da situação e, por exemplo, Rewitzky [34] defende que um sistema de geração automática de questões e testes permite, não só que os alunos dominem os vários assuntos em estudo, mas também a criação de testes diferentes para cada aluno, evitando, dessa forma, a cópia de respostas pelos alunos. Mais ainda, permite uma correção e análise rica dos testes e exames, providenciando aos avaliadores, não só uma economia de tempo, mas também uma leitura do respetivo percurso/perfil de resolução, permitindo evidenciar as matérias em que os alunos tenham maior dificuldade.

1.1 Geração automática de exercícios

De um modo geral, a resolução de exercícios consiste em estabelecer um método f para determinar os *pedidos* p a partir dos *dados* d:

$$d \xrightarrow{f} p.$$

Não haveria ciência se não procurarmos estruturas em d, p e f: os problemas seriam todos diferentes, não haveria lugar a transportar técnicas e métodos de uns problemas para outros *semelhantes*.

A geração de exercícios na área de Programação e de Linguagens de Programação precisa de uma sólida base matemática formulada em termos de gramáticas generativas visto que os exercícios nesta área têm uma rica estrutura matemática e a sua geração envolve mecanismos lógicos/algébricos.

1.1.1 Estado da arte

O desenvolvimento de sistemas de geração automática de exercícios e sistemas de ensino assistido por computador, tornou-se um campo de grande atividade quer no mundo [10, 12, 13, 18, 34, 40], quer em Portugal [21, 22, 33, 42, 43, 44, 45, 46].

Este domínio tem sido palco de activa investigação por diversas escolas. A título de exemplo, na Universidade do Minho foi acumulada alguma experiência na área de *e-learning* ligado à Matemática referindo-se por exemplo:

- utilização do sistema Maple TA no processo de ensino e avaliação [33];
- nos últimos anos foi desenvolvida por J. João Almeida uma linguagem de programação PASSAROLA orientada para criação e correção automática de exercícios [4];
- foram feitos primeiros passos no desenvolvimento da teoria geral de geração de exercícios [3];

- sob orientação do Pedro Rosário (Escola de Psicologia da UM) foram desenvolvidos vários sistemas de ensino de Matemática com utilização do computador [30, 1];

- no Departamento de Informática (Escola de Engenharia da UM) tem sido usado avaliação automática de códigos simples em disciplinas de introdução à programação bem como em torneios de programação (TIUP, em colaboração com outras universidades do país).

Tudo isto criou bases para uma investigação científica no domínio dos sistemas de ensino personalizado na área dos fundamentos da programação [11, 39] com geração automática de exercícios, no sentido de desenvolvimento de princípios gerais de elaboração de exercícios com estrutura não trivial e correção automática.

1.1.2 Objetivos

Esta tese tem como objetivo principal o desenvolvimento de bases teóricas para a elaboração de sistemas de *e-learning* centrado no domínio de Programação, nomeadamente em gramáticas e linguagens independentes de contexto.

De um modo geral, um sistema de *e-learning* é composto por um gerador automático de exercícios, um verificador de respostas e uma estrutura personalizada de plano de estudos.

Esta tese centrou-se na geração de exercícios, nomeadamente no desenvolvimento de algoritmos e gramáticas geradoras necessários para a geração de exercícios, e no desenvolvimento de estratégias de verificação automática da resposta dada a determinados problemas.

A escolha da área das linguagens e gramáticas prendeu-se com:

- as gramáticas são ferramentas essenciais para uso e criação de linguagens de programação;

- a capacidade de escrever boas gramáticas, capazes de elegantemente definir uma linguagem é algo difícil de adquirir por parte dos estudantes;

- as gramáticas ajudam a organizar o conhecimento ligado a uma língua, constituindo um mecanismo de organizar o pensamento de um programador.

1.2 Geração de exercícios – metagramáticas

Ligado ao processo de geração de exercícios, estamos interessados nas diversas etapas:

- criação dos enunciados,
- criação das resoluções,
- processo de verificação automática.

Consideremos um exemplo simples de exercício de escrita de gramáticas: *Escreva a gramática que gere um ou mais* ***a*** *seguido de qualquer número de* ***b*** *ou* ***c****.*

Este exemplo pode facilmente ser generalizado para uma família de casos análogos, genericamente estruturados como uma linguagem composta por duas partes:

- uma primeira parte (P1) em que se peçam sequências de **a** com diferentes restrições,
- uma parte (P2) em que se "misturem"**b** e **c** de diversos modos.

Esta generalização leva à introdução do conceito de metagramática geratriz de gramáticas (e obviamente do respectivo enunciado). Seguidamente apresentamos um exemplo de uma metagramática que descreve a generalização proposta:

$$
\begin{array}{lcl}
\boxed{\text{G}} & \rightarrow & S : p1\ p2\ ;\ \boxed{\text{P1}}\ \boxed{\text{P2}}\ ;\ \{\text{Escreva a gramática que gere \$P1 seguido de \$P2.}\} \\
 & . & \\
\boxed{\text{P1}} & \rightarrow & p1 : \varepsilon \mid a\ p1\ \{\text{zero ou mais 'a'}\} \\
 & \| & p1 : a \mid a\ p1\ \{\text{um ou mais 'a'}\} \\
 & \| & p1 : \varepsilon \mid a\ \{\text{'a' opcional}\} \\
 & . & \\
\boxed{\text{P2}} & \rightarrow & p2 : b\ c\ \{\text{um 'b' e um 'c'}\} \\
 & \| & p2 : b \mid c\ \{\text{um 'b' ou um 'c'}\} \\
 & \| & p2 : b\ p2 \mid c\ p2 \mid \varepsilon\ \{\text{qualquer número de 'b' ou 'c' (eventualmente zero)}\} \\
 & . &
\end{array}
$$

De um modo simplificado,

- os elementos com retângulos são os não terminais da metagramática (que expandem para partes da gerada gramática);
- os símbolos '||' denotam alternativas de produções da metagramática

- os *templates* incluídos entre '{...}' definem o enunciado (após a expansão das variáveis '$P').

Uma explicação detalhada deste formalismo é apresentada no capítulo 3, onde se demonstra como as metagramáticas podem ser usadas para geração de exercícios e respetivas resoluções.

Esta metagramática definiu 3 variantes para cada uma das partes (P1, P2) gerando portanto 9 diferentes exercícios (9 enunciados e 9 resoluções).

1.3 Verificação automática

Na área dos exercícios sobre linguagens e gramáticas, há necessidade de contemplar uma variada gama de verificações tais como:

- ver se uma frase pertence a uma linguagem;
- ver se uma árvore de derivação está correta;
- geração de frases;
- ver se duas gramáticas são equivalentes ou não.

A verificação de ser uma frase válida numa gramática, baseia-se na geração de um parser associado à gramática, usando a ferramenta Bison, sendo este posteriormente usado para validar sintaticamente as frases (ver cap. 5).

A verificação de árvore de derivação, é sumariamente abordada no exemplo da secção 3.1.4.

A geração de frases válidas numa gramática apresenta vários enunciados possíveis. Por exemplo, gerar todas as frases com comprimento menor que N, gerar todas as frases cuja árvore tenha uma profundidade menor que N, gerar uma frase aleatória. Algumas das variantes são simples para os casos gerais, mas têm uma elevada complexidade para certas situações.

Dos pontos referidos o mais complexo é a comparação de gramáticas – habitualmente descrito como problema indecidível para o caso de linguagens independentes de contexto.

Consideremos a linguagem correspondente à definição de listas generalizadas de letras, incluindo frases como

```
( a a ( a () (a a)))
```

Um gramática possível para descrever esta linguagem é

```
S  →  ( L )
   ;
L  →  L F
   |  ε
   ;
F  →  S
   |  a
   ;
```

Pretendemos saber se uma outra gramática submetida por um estudante é ou não equivalente à gramática esperada:

```
B  →  ( K )
   ;
K  →  a K
   |  B K
   |  ε
   ;
```

Como é fácil de notar, as gramáticas, os axiomas e os símbolos não-terminais são diferentes. Porém mesmo alguém experiente, tem de fazer algum esforço para concluir se as linguagens associadas são ou não coincidentes (neste caso são).

Para evitar confusões com as notações, vamos substituir os símbolos terminais '(' e ')' por 'b' e 'c', ficando as duas gramáticas com o seguinte aspecto:

```
S  →  b L c
   ;
L  →  L F
   |  ε
   ;
F  →  S
   |  a
   ;
```

e

```
B  →  b K c
   ;
K  →  a K
   |  B K
   |  ε
   ;
```

O conjunto das frases válidas desta linguagem é obviamente infinito.

É bem conhecido que o triplo de (concatenação, reunião, linguagens), tem muitas propriedade análogas às de um anel, faltando-lhe a existência de

elemento inverso da reunião. Existe portanto um parentesco natural entre as linguagens definidas por gramáticas e qualquer anel, nomeadamente com os anéis comutativos dos números inteiros, números racionais, números reais, números complexos com as multiplicações e adições habituais e ainda com os anéis (não comutativos) de matrizes e respectivas multiplicações e adições.

As linguagens habituais são normalmente infinitas, e não têm uma representação direta em computador (um conjunto infinito de palavras ocuparia uma quantidade infinita de memória). Ao contrário, tipos como racionais, reais, complexos, matrizes têm representações em computador mais viáveis. Com uma escolha cuidada dos seus valores temos somas infinitas a darem resultados finitos e perfeitamente representáveis.

Considere-se a seguinte linguagem:

$$L = \{\varepsilon, a, aa, \ldots\}$$

Se considerarmos uma substituição $a \to \frac{1}{2}$ podemos fazer corresponder a esta linguagem a seguinte série:

$$L = 1 + \frac{1}{2} + \frac{1}{4} + \ldots = 2.$$

O parentesco com anéis comutativos é menos interessante do que aquele que existe com anéis não comutativos, já que temos interesse em não acrescentar propriedades que tornem iguais linguagens diferentes. Isto leva-nos a privilegiar anéis de matrizes.

Normalmente no desenho de linguagens, pretende-se evitar a ambiguidade, já que elas conduzem a futuros problemas semânticos. Se no entanto lidarmos com linguagens ambíguas, os modelos habituais de reunião "esconde"essa ambiguidade devido à idempotência ($\{a\} \cup \{a\} = \{a\}$). Ao contrário, os anéis não gozam de idempotência ($a + a \neq a$, $a \neq 0$). Deste ponto de vista, considera-se diferentes linguagens que tenham diferentes graus de ambiguidade [1].

A comparação de gramáticas independentes de contexto é um problema indecidível. Isto é verdade quando a comparação de gramáticas é considerada como um problema algébrico. Nesta tese esta comparação é considerada como problema de análise; deste modo o problema é resolúvel.

Dum modo simplificado, transformamos o domínio algébrico das gramáticas da seguinte maneira:

- cada símbolo terminal é substituído por uma matriz ($\mathcal{N} \times \mathcal{N}$);

[1] Este comportamento na generalidade dos casos será desejável, mas nalguns casos pode não ser o pretendido.

- a reunião e concatenação de linguagens são transformadas na soma e multiplicação de matrizes;
- uma gramática é transformada num sistema de equações não lineares matriciais, que é resolvido numericamente.

Deste modo duas linguagens serão iguais se as matrizes correspondentes ao axioma coincidem para as diversas substituições de terminais por matrizes $(\mathcal{N} \times \mathcal{N})$ [2].

1.4 Trabalho experimental

Juntamente com o trabalho matemático sentimos necessidade de criar um conjunto de ferramentas e protótipos que nos permitissem experimentar e validar os conceitos propostos com casos reais. Optamos por agrupar as ferramentas construídas num *toolkit* (ver cap. 5).

Nalguns pontos do trabalho foi necessário recorrer a testes com grandes quantidades de cálculos. Para tal recorremos ao uso de um cluster.

A criação e utilização das ferramentas e protótipos, contribui muito para a clara especificação dos problemas e das restrições de aplicabilidade dos métodos desenvolvidos.

1.5 Resultados principais da tese

Os resultados principais apresentados nesta tese são:

- Formalismo de metagramáticas para geração de exercícios;
- Teoremas de distinguibilidade para gramáticas independentes de contexto;
- Algoritmos de comparação de gramáticas independentes de contexto;
- Análise experimental da aplicabilidade de algoritmos de comparação de gramáticas independentes de contexto;
- Desenvolvimento de um conjunto de ferramentas de processamento de gramáticas.

Os resultados principais encontram-se em [5, 6, 7].

[2] Os detalhes deste processo são discutidos no capítulo 4.

1.6 Organização do trabalho

O trabalho está organizado do seguinte modo: No capítulo 2, apresenta-se um conjunto de conceitos básicos necessários para a leitura desta tese. No capítulo 3, descreve-se o formalismo de metagramáticas para geração de exercícios. O capítulo 4 inclui os teoremas de distinguibilidade para gramáticas independentes de contexto e os algoritmos de comparação. No capítulo 5, descreve-se o conjunto de ferramentas de processamento de gramáticas. Finalmente o capítulo 6 inclui as conclusões e planos de trabalho futuro.

Capítulo 2

Conceitos básicos

Neste capítulo apresenta-se sumariamente um conjunto de tópicos cujo conhecimento é necessário para a leitura do resto da tese.

2.1 Linguagens e gramáticas

Seja V um alfabeto. O conjunto de palavras sobre V vamos designar por $W(V)$. Uma linguagem sobre o alfabeto terminal V_T, é um conjunto de palavras válidas $W(V_T) = V_T^*$.

Classicamente, uma gramática (independente de contexto) é um quádruplo $G = (V_N, V_T, P, S)$ onde:

- V_N é um conjunto não vazio e finito dos símbolos não-terminais;
- V_T é um conjunto não vazio e finito dos símbolos terminais;
- P é um conjunto de produções, onde cada produção é formada por (lhs, rhs), sendo $lhs \in V_N$ e $rhs \in (V_T \cup V_N)^*$;
- $S \in V_N$ é o símbolo inicial ou axioma.

2.1.1 Séries formais de potências

Qualquer linguagem independente de contexto pode ser definida em termos de uma série de potências formal com variáveis associativas, mas não comutativas [35, 36]. Seja V_T o alfabeto terminal, $W(V_T)$ o conjunto de palavras sobre V_T, e Z_+ o conjunto de inteiros não-negativos. Uma aplicação $\phi : W(V_T) \to Z_+$ define uma série de potências formal

$$S = \sum_{P \in W(V_T)} \phi(P)P. \tag{2.1}$$

2.1.2 Sistemas de equações correspondentes a uma gramática

Sabe-se que a qualquer gramática independente de contexto corresponde um sistema de equações não-lineares [35, 36] que permite obter as respectivas séries de potências formais através de iterações sucessivas. Os termos da série são as palavras da respectiva linguagem. Seja X_i, $i = \overline{0,m}$, não-terminais de uma gramática independente de contexto e sejam P_j^i, $i = \overline{0,m}$, $j = \overline{1,l_i}$, as palavras que aparecem no lado direito das produções com os lados esquerdos X_i. O sistema de equações correspondente à gramática tem a forma

$$\begin{aligned} X_1 &= P_1^1 + \ldots + P_{l_1}^1, \\ &\vdots \\ X_m &= P_1^m + \ldots + P_{l_m}^m. \end{aligned}$$

2.1.3 Forma normal de Chomsky

Uma gramática independente de contexto está na forma normal de Chomsky se todas as suas regras de produção são da forma:

$$\begin{aligned} &A \to BC \text{ ou} \\ &A \to a \text{ ou} \\ &S \to \varepsilon \end{aligned}$$

onde $A, B, C \in V_N$, $a \in V_T$ e ε é o símbolo vazio. Além disso, nem B nem C podem ser a variável inicial.

Uma gramática independente de contexto está na forma normal reduzida de Chomsky se todas as suas regras de produção são da forma:

$$\begin{aligned} &A \to BC \text{ ou} \\ &A \to a \end{aligned}$$

Toda a gramática independente de contexto pode ser transformada numa gramática equivalente que esteja na forma normal (normal reduzida) de Chomsky.

2.1.4 Forma normal de Greibach

Uma gramática independente de contexto está na forma normal de Greibach quando todas as suas produções são da forma:

$$A \to a\alpha$$

onde $A \in V_N$, $a \in V_T$ e $\alpha \in W(V_N)$. Toda a gramática independente de contexto pode ser transformada numa gramática equivalente que está na forma normal de Greibach.

2.2 Aplicações lineares

Sejam X e Y espaços normados de dimensão finita sobre o corpo dos números reais R. Designemos por $\mathcal{L}(X, Y)$ o conjunto das aplicações lineares de X em Y. Sejam $A_1, A_2 \in \mathcal{L}(X, Y)$ e $\alpha \in R$. Definamos as operações lineares no conjunto $\mathcal{L}(X, Y)$ de modo seguinte:

$$(A_1 + A_2)(x) = A_1x + A_2x, \quad (\alpha A)(x) = \alpha Ax.$$

É fácil ver que, com estas operações de adição e multiplicação por um número, o conjunto $\mathcal{L}(X, Y)$ torna-se um espaço linear.

Consideremos uma aplicação linear $A \in \mathcal{L}(X, Y)$. Como X e Y são de dimensão finita, o valor

$$\|A\|_{\mathcal{L}} = \sup_{\{x \in X,\ \|x\|_X = 1\}} \|Ax\|_Y$$

é finito. É fácil mostrar que $\|A\|_{\mathcal{L}}$ é uma norma no espaço $\mathcal{L}(X, Y)$. Obviamente, se $A \in \mathcal{L}(X, X)$ e $x \in X$, então

$$\|Ax\| \leq \|A\|\|x\|.$$

Se $A, B \in \mathcal{L}(X, X)$ então

$$\|AB\| \leq \|A\|\|B\|.$$

Estas propriedades da norma de uma aplicação linear serão muito úteis no Capítulo 4.

2.3 Teorema do ponto fixo

Recordemos um resultado muito importante sobre existência de pontos fixos de aplicações contractantes.

Teorema 1. *Sejam $C \subset R^n$ um conjunto fechado e $F : C \to C$ uma aplicação contractante, i.e. uma aplicação que verifica a condição*

$$\|F(X_1) - F(X_2)\| \leq q\|X_1 - X_2\|,$$

onde $0 < q < 1$ *e* X_1 *e* X_2 *são quaisquer pontos de* C*. Então existe um único ponto (ponto fixo)* $\hat{X} \in C$ *tal que* $F(\hat{X}) = \hat{X}$*. Além disso, para qualquer ponto* $X_0 \in C$*, a sucessão* $X_{k+1} = F(X_k)$*,* $k = 0, 1, 2, \ldots$*, converge para* $\hat{X}$*.*

Este teorema será utilizado no Capítulo 4.

Capítulo 3

Geração de exercícios de gramáticas/linguagens

É bem sabido que um sólido conhecimento dos formalismos e ferramentas de especificação de linguagem é muito importante na ciência da computação [1]. Currículos de ciência da computação sempre incluem módulos nestes assuntos sensíveis e podemos encontrar vários bons exercícios sobre este tema [2].

Quando lidamos com a geração automática de exercícios de especificação de linguagem (capazes de produzir conjuntos de exercícios semelhantes), precisamos processar os componentes básicos de especificação de linguagens e gerá-los, transformá-los e compô-los para produzir os enunciados, as soluções e funções de validação/verificação de exercícios. Isso não é fácil e claramente precisa de métodos e ferramentas de suporte.

Neste capítulo discutimos os componentes típicos de um exercício de especificação de linguagem e apresentamos a linguagem **Mgbeg** desenvolvida para criar exercícios neste domínio e alguns exemplos.

Componentes de exercícios de linguagem

Um exercício típico sobre especificação básica de linguagens, envolve:

- uma descrição em linguagem natural;
- uma especificação de linguagem (expressão regular, linguagem independente de contexto, autómatos);
- exemplos de frases válidas;

[1] O mesmo conhecimento é importante para a linguística aplicada e outras áreas relacionadas.

- perguntas se algumas frases pertencem à linguagem;
- conversão entre formatos de especificação de linguagens;
- desenho de árvores de derivação de uma frase;
- semântica de frases (este tópico não é abordado neste trabalho).

Os exercícios desenvolvidos envolvem a manipulação de um conjunto de componentes de linguagem. É claro que os diversos componentes não são independentes entre si e portanto não podem ser gerados separadamente.

Linguagem Mgbeg

Seguidamente apresenta-se a linguagem **Mgbeg** usada para especificar a geração de exercícios que permite criar uma grande quantidade de exercícios diferentes a partir de um esquema de base gramatical.

O processo de geração de exercícios de especificação de linguagem pode ser visto como:

- definição de um conjunto de gramáticas semelhantes - esta definição é feita usando uma meta-gramática;
- enriquecimento da meta-gramática com um conjunto de regras de atributos e modelos capazes de calcular formas alternativas e objetos derivados (Exemplo destes atributos são: um enunciado em linguagem natural, exemplos de frases válidas, distratores, e estruturas de dados para ajudar a avaliar as respostas do aluno);
- uso de um gerador aleatório top-down (cuja definição e detalhes serão apresentados na secção 3.2) para escolher uma gramática e atributos associados;
- utilização dessas informações para preencher os modelos de exercícios.

Uma abordagem inicial para a geração de exercícios consiste na construção de uma base de exercícios da qual sortearemos as variantes pretendidas, e pode ser esquematicamente representada pelo seguinte algoritmo:

```
type exercise-db: (exer_stat, result)*
```

Algorithm 1: generate(e: exercise-db)

begin
    $x \leftarrow choice(e)$
    return x

Uma abordagem mais sofisticada baseada em esquemas de exercícios gerais pode ser descrita da seguinte forma:

```
type exercise-db: exer-sch*
     exer-sch: (template, table)
     table: (vars, result)*
```

Algorithm 2: generate(e: exercise-db)

begin
- $x \leftarrow choice(e)$
- $row \leftarrow choice(x.table)$
- $exer\_stat \leftarrow x.template(row.vars)$
- return $(exer\_stat, row.result)$

Esta abordagem ajuda a compreender melhor a estrutura do exercício e a gerar um grande número de variantes.

A linguagem **Mgbeg**, apresentada neste capítulo, generaliza a abordagem acima apresentada para *templates*, variáveis e resultados multi-nível. A generalização dos esquemas de exercício é sintaticamente descrita em termos de uma meta-gramática; de forma simplificada, as escolhas aleatórias correspondem a frases geradas pela meta-gramática.

O capítulo está organizado da seguinte forma. Na secção 3.1 discutimos o conceito de meta-gramática, sua estrutura e atributos. Na secção 3.2 descrevemos brevemente o módulo auxiliar **Mgbeg**, usado na implementação da linguagem **Mgbeg**. Finalmente, apresentamos algumas conclusões.

3.1 Meta-gramatica

Como foi anteriormente dito, uma meta-gramática (MG) é uma gramática usada para descrever e gerar gramáticas. O conjunto de todas as cadeias de derivação possíveis, define o grupo de exercícios coberto pela meta-gramática [29]. Notemos que o termo usado "meta-gramática"difere das gramáticas Meta-S (§-Grammars) correspondentes às gramáticas adaptativas [25].

3.1.1 Estrutura de uma meta-gramática

Definição: Meta-gramática é um quádruplo $MG = (MN, MT, MP, MS)$ sendo:

- MN é um conjunto não vazio e finito dos símbolos não-terminais;

- MT é um conjunto não vazio e finito dos símbolos terminais;
- MP é um conjunto de produções, onde cada produção é formada por (lhs, rhs), sendo $lhs \in MN$ e $rhs \in (MT \cup MN)^*$;
- $MS \in MN$ é o símbolo inicial ou axioma.

Para gerar exercícios, a meta-gramática, além da geração de uma gramática, também gera informação atributiva guiada pelas regras de atributos definidas em cada produção.

Seguidamente apresenta-se a notação utilizada:

Notação	**Descrição**
MN	identificação de não-terminal da meta-gramática
‖	separação de produções da meta-gramática
”.”	finalização da(s) produção(ões) de um lhs
$\$X$	valor semântico do símbolo X
{*enunciado*}	*template* de enunciado

3.1.2 Exemplo

No exemplo apresentado na tabela 3.1, definimos uma MG para gerar exercícios de listas e listas generalizadas de números inteiros ou letras.

Neste exemplo de MG temos que o $MN = \{MS,\ K,\ Sep,\ T\}$ é o conjunto dos símbolos não terminais e $MT = \{S,\ E,\ L,\ FL,\ ALFA,\ INT,\ :,\ ;,\ -,\ |\}$ é o conjunto de símbolos terminais. A fim de ter uma leitura mais clara, em meta-gramáticas, escrevemos não-terminais dentro de caixas. Cada frase gerada pela meta-gramática é uma gramática e seus atributos associados (neste caso um único atributo - o *enunciado*).

Na Tabela 3.2 apresentamos uma cadeia de derivação e uma árvore sintática para a frase ”S : E , S | E ; E : INT ;”(Correspondente à gramática da linguagem de listas de inteiros (INT) separados por vírgulas).

Agora vamos criar uma atividade de aprendizagem usando os componentes criados pela meta-gramática acima. Esta atividade é composta de duas etapas:

1. Usando a meta-gramática, criamos duas gramáticas (por omissão as gramáticas são armazenadas no campo ”g”) e atributos associados (neste caso, o único atributo é enunciado – campo ”e”)

[MS]	→	S : [K] ; {Considera a linguagem de $K. Crie uma gramática que gere esta linguagem.}
	.	
[K]	→	E [Sep] S | E ; E : [T] {uma lista de $T por $Sep}
	‖	(L) ; L : L FL | ; FL : S | [T] {lista generalizadas de $T por parênteses }
	.	
[Sep]	→	, {virgula}
	‖	/ {barra}
	‖	- {traço}
	.	
[T]	→	INT {inteiros (INT) separados}
	‖	ALFA {letras (ALFA) separadas}
	.	

Tabela 3.1: Exemplo de meta-gramática de lista ou lista generalizada de inteiros ou letras.

2. Usando estes *componentes de linguagem*, preenchemos as lacunas do modelo de atividade.

Consideremos o seguinte código de definição de atividade:

```
a = mgbeg("lists.mg", exercise=2, example=3)
                  // a: Conjunto de componentes (2 gramáticas e 3 frases exemplo)
#question
    #a[0].e                       // enunciado de exercício da primeira gramática
    Exemplos de frases válidas:
    . #a[0].example[0]            // frase exemplo
    . #a[0].example[1]
    . #a[0].example[2]
#result
    #a[0].g                       // a gramática criada
#validation
    gramequiv(#1, #a[0].g)        // gramática submetida = primeira gramática
#question
    (a) #a[1].e
    (b) Escreva uma frase válida.
#result
    (a) #a[1].g
    (b) Exemplos de frases válidas:
      . #a[1].example[0]
```

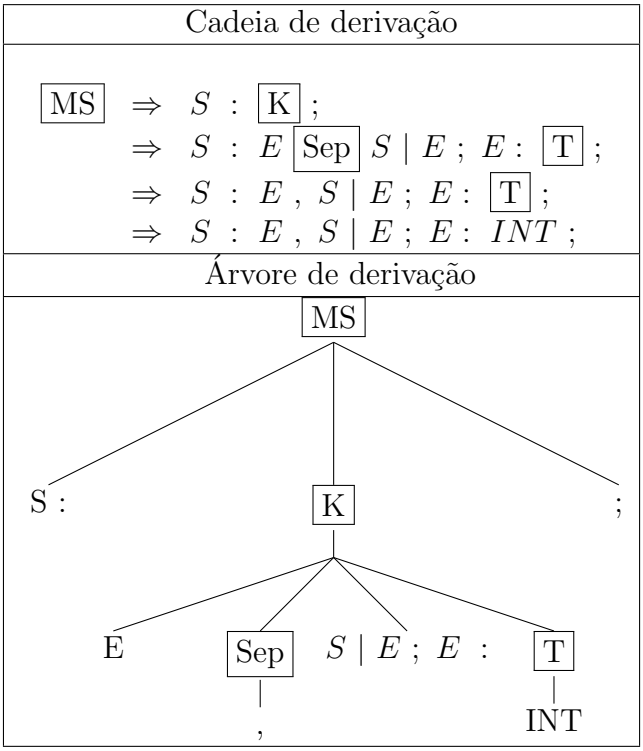


Tabela 3.2: Árvore de derivação da sequência "S : E , S | E ; E : INT ;".

```
      . #a[1].example[1]
      . #a[1].example[2]
#validation
    (a)gramequiv(#1, #a[1].g);   // gramática submetida = segunda gramática
    (b)validsent(#a[1].g, #1)    // é uma frase válida na gramática
```

Após o processamento do código anterior, são criados vários documentos, nomeadamente:

(i) O enunciado

1. Considere a linguagem de listas de letras (ALFA) separadas por um traço. Crie uma gramática que gere esta linguagem.

   Exemplos de frases válidas:
   v - a
   d - o
   h - k - g

2. (a) Considere a linguagem das listas generalizadas de letras (ALFA) separadas por parênteses. Crie uma gramática que gere esta linguagem.

   (b) Escreva uma frase válida.

(ii) A resolução

1. S : E - S | E ;
   E : ALFA ;

2. (a) S : (L) ;
   L : L FL | ε ;
   FL : S | ALFA ;

   (b) Exemplos de frases válidas:
   ( )
   ((a) ((b (d)) ( )))
   ((( ) b e) (b a f) x ( ))

(iii) O script de verificação (não incluído)

3.1.3 Atributos da meta-gramática

A construção de meta-exercícios (modelos para gerar exercícios) é um processo complexo. A fragmentação dos exercícios em partes a combinar mais tarde, gerando enunciados elegantes são problemas desafiadores.

Para gerar gramáticas com atributos associados (componentes de linguagem), as produções meta-gramáticas do **Mgbeg** podem adicionalmente usar regras de atributos. Estão contemplados os seguintes formatos de regras de atributos:

- `id:{...template with variables}` – para atribuir ao atributo *id* o valor do *template* depois de expandir as variáveis;
- `id:!{...expression with variables}` – para atribuir ao atributo *id* o resultado do cálculo da expressão após a expansão das variáveis;
- `{template with variables}` e `!{...expression with vars.}` – (Idêntico ao anterior) - para atribuir ao atributo padrão *e* (enunciado).

Este tipo de regras foram usadas no exemplo anterior.

Nas regras de atributos (*templates* e expressões) as variáveis referem-se aos valores dos atributos de símbolos presentes no *rhs* da produção. Os seguintes formatos são suportados:

- `$A.id` – atributo *id* do símbolo não-terminal A
- `$A.g` – atributo g (a gramática) do não-terminal A
- `$A` – atributo e (enunciado) do não-terminal A

3.1.4 Exemplo com atributos complexos

Consideremos a seguinte gramática:

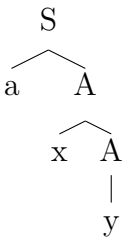
$$\begin{array}{rcl} S & \rightarrow & aA \\ & | & b \\ & ; & \\ A & \rightarrow & xA \\ & | & y \\ & ; & \end{array}$$

Um exercício comum é: *Escrever uma frase válida com pelo menos 3 símbolos e apresentar a correspondente árvore sintática.*

Para desenhar árvores sintáticas, várias ferramentas (como rsyntaxtree [23], phpsyntaxtree [20], ou qtree) usam a notação de parêntesis rectangulares para descrever a estrutura da árvore:

```
[S a [A x [A y ] ] ]
```

para a árvore



O exemplo a seguir mostra como, usando **Mgbeg**, é possível construir, juntamente com a gramática acima, uma expressão regular correspondente e ainda uma gramática dos desenhos de árvore válidos.

$$
\begin{array}{lcl}
\boxed{\text{G}} & \rightarrow & S : a\ A \mid b\ ; \boxed{\text{P}} \\
 & & re : \{a\ \$P.re + b\} \\
 & & stg : \{S1 : [S\ a\ A1] \mid [S\ b]\ ;\ \$P.stg\} \\
 & \| & S : A\ a \mid b\ ; \boxed{\text{P}} \\
 & & re : \{\$P.re\ a + b\} \\
 & & stg : \{S1 : [S\ A1\ a] \mid [S\ b]\ ; \$P.stg\} \\
 & . & \\
\boxed{\text{P}} & \rightarrow & A : x\ A \mid y \\
 & & re : \{x^*\ y\} \\
 & & stg : \{A1 : [A\ x\ A1] \mid [A\ y]\} \\
 & \| & A : A\ x \mid y \\
 & & re : \{y\ x^*\} \\
 & & stg : \{A1 : [A\ A1\ x] \mid [A\ y]\} \\
 & \| & A : x\ A \mid x\ y \\
 & & re : \{x^*\ x\ y\} \\
 & & stg : \{A1 : [A\ x\ A1] \mid [A\ x\ y]\} \\
 & . &
\end{array}
$$

Como resultado da geração baseada neste meta-gramática temos 6 tuplos da forma:

```
g  : S → A a | b ; A → x A | x y      // Gramática
re : x*xya + b                        // Expressão regular
stg: S1→[S A1 a] | [S b]; A1→[A x A1] | [A x y]  // Árvore
```

Este exemplo ilustra o uso de atributos complexos para calcular componentes para o enunciado e avaliação. (No processo de avaliação, verificamos se a árvore sintática submetida pertence à linguagem gerada pela gramática de desenho de árvore construída).

3.2 Módulo Mgbeg

Ficou claro desde o início que era crucial ter um rico *kit* de ferramentas com funções capazes de manipular gramáticas e outros componentes de linguagem.

Abaixo apresentamos algumas das funções mais utilizadas do módulo **Mgbeg**.

gramequiv: grammar, grammar → bool — dadas duas gramáticas, a função determina se elas descrevem a mesma linguagem. (Várias outras funções de equivalência gramatical são possíveis [15].)

validsent: grammar, sentence → bool — verifica se a frase é válida.

grampp: grammar → LaTeX — formata a gramática em LaTeX.

bisongram: grammar → bison-parser — gera e compila um analisador Bison [19] simples para a gramática.

gramgen: grammar → sentence — gera exemplos aleatórios de frases válidas.

mgramgen: metagrammar → (id → value)* — esta função é um gerador top-down. Retorna uma gramática, enunciado e informações de atributos relacionadas. Descrito no Alg. 3.

mgbeg: metagrammar, options → seq of (id → value)* — gera uma ou mais gramáticas, atributos e frases exemplo.

Algorithm 3: mgramgen(mg,ax): Geração de gramática e enunciado

Input: mg: meta-grammar
Input: ax: $axiom \in NT$
Output: r: mapping id $\rightarrow$ (grammar, statement or language component)

```
begin
    if ax ∈ T then                                  //geração de simbolo terminal
        return (g → ax)
    if ax ∈ NT then
        r[g] ←' '                                   // inicialização da gramática
        rhss ← mg[ax]
        (rhs, attrules) ← choice(rhss)              //seleção aleatória de produção
        for s ∈ rhs do
            aux ← gen(mg, s)
            r['g'] ← r['g'] + aux['g']
        for (id : t) ∈ attrules do
            if t is !{...} then                     //template escrito em Perl
                aux ← template_expand_vars(t, r)
                aux ← eval(aux)
            if t is {...} then                      //template normal
                aux ← template_expand_vars(t, r)
            r[id] ← aux
    return r
```

3.3 Conclusão

Neste capítulo apresentamos um algoritmo para a geração de exercícios no domínio das linguagens formais, para gramáticas independentes de contexto. O algoritmo faz uso de meta-gramáticas. As principais vantagens desta abordagem são:

- O uso de meta-gramáticas ajuda a estruturar o exercício;
- Os atributos organizam a especificação de modo declarativo de alto nível;
- O uso de meta-gramática permite que se trate de forma unificada todos os elementos do exercício, tais como enunciado, exemplos, gramáticas,

expressões regulares, árvores sintáticas, componentes de avaliação.

Capítulo 4

Comparação de gramáticas independentes de contexto

Uma a linguagem sobre o alfabeto terminal V_T, é um conjunto de palavras válidas $W(V_T) = V_T^*$. Como este conjunto é geralmente infinito, é necessário usar gramáticas como um mecanismo para a definição das linguagens. Vamos considerar somente as gramáticas independentes de contexto, abrangendo as linguagens independentes de contexto. A capacidade de escrever gramática correta é uma tarefa essencial na ciência da computação (usada, por exemplo, na criação de linguagens de programação, compiladores, etc.). A avaliação das respostas dos alunos em ciência da computação é uma atividade muito demorada. A avaliação assistida por computador é uma forma natural de reduzir o tempo gasto pelos professores na sua tarefa de avaliação (ver, por exemplo, [9, 8]). Este capítulo trata da avaliação na teoria das gramáticas independentes de contexto. Seu principal objetivo é criar uma base teórica para algoritmos que permitam decidir se duas gramáticas independentes de contexto são equivalentes ou não.

É bem conhecido que a equivalência de duas gramáticas independentes de contexto é um problema indecidível [35]. Isso é verdade se considerarmos o problema como um problema algébrico. Neste capítulo mostramos que o problema admite uma solução se for considerado como um problema de análise matemática. Tal situação não é nova. Por exemplo, o resultado do trabalho [48] pode ser interpretado como uma impossibilidade de demonstração o último teorema de Fermat usando meios aritméticos. Por outro lado, o famoso artigo de Wiles [47] contém a demonstração, mas é baseada em ideias de matemática contínua.

O problema da equivalência de gramáticas independentes de contexto foi objeto de numerosos estudos [15, 16]. Por exemplo, o problema foi resolvido quando a equivalência é entendida em sentido estrutural [31]. Além

disso foram desenvolvidos alguns algoritmos práticos para a verificação de equivalência de gramáticas (veja [27, 28], e suas referências).

A fim de apresentar a metodologia adotada neste capítulo, consideremos o seguinte simples exemplo. Seja L a linguagem

$$L = c, ab, acb, accb, acccb, ...$$

De acordo com [36] podemos escrever uma série de potências formal

$$S = c + ab + acb + accb + acccb + \dots \tag{4.1}$$

correspondente a esta linguagem. A linguagem L pode ser gerada pela gramática

$$\begin{array}{rcl} S & \rightarrow & aAb \\ & | & c \\ & ; & \\ A & \rightarrow & cA \\ & | & \varepsilon \\ & ; & \end{array}$$

O seguinte sistema de equações formais correspondentes à gramática é

$$S = aAb + c, \tag{4.2}$$
$$A = cA + \epsilon. \tag{4.3}$$

Aplicando formalmente o método iterativo a este sistema obtemos séries (4.1). Abaixo definimos uma transformação que atribui um valor matricial à série de potências formal (4.1). Isto significa que,

- cada letra terminal a, b, c, é substituída por matriz $(\mathcal{N}\times\mathcal{N})$ tornando-se μ_a, μ_b, μ_c;
- os símbolos não-terminais S e A são substituídos por matrizes $(\mathcal{N}\times\mathcal{N})$ designadas por $S(\mu)$ e $A(\mu)$;
- a soma formal e o produto são substituídos pelas soma e produto de matrizes;
- a palavra vazia ϵ é substituída pela matriz identidade $(\mathcal{N}\times\mathcal{N})$.

Então, a S corresponde a matriz $S(\mu) = S(\mu_a, \mu_b, \mu_c)$ calculada como a soma da série matricial

$$S(\mu) = \mu_c + \mu_a\mu_b + \mu_a\mu_c\mu_b + \mu_a\mu_c\mu_c\mu_b + \mu_a\mu_c\mu_c\mu_c\mu_b + ...$$

Para calcular efetivamente esta soma resolvemos numericamente o sistema de equações matriciais

$$S(\mu) = \mu_a A(\mu)\mu_b + \mu_c,$$
$$A(\mu) = \mu_c A(\mu) + I,$$

obtido aplicando a transformada ao sistema formal (4.2) e (4.3). Da mesma forma, no caso geral de uma gramática com o alfabeto terminal $V_T = \{a_1, ..., a_n\}$, pode-se calcular a matriz $S(\mu) = S(\mu_{a_1}, \ldots, \mu_{a_n})$. O principal resultado demonstrado neste capítulo (Teorema de Distinguibilidade I) mostra que se duas linguagens L_1 e L_2 geradas por gramáticas independentes de contexto são diferentes, então existe uma substituição matricial $\mu_{a_1}, \ldots, \mu_{a_n}$ tal que

$$S_1(\mu_{a_1}, \ldots, \mu_{a_n}) \neq S_2(\mu_{a_1}, \ldots, \mu_{a_n})$$

(Note que as linguagens com diferentes ambiguidades consideramos como linguagens diferentes.) Esta propriedade permite construir ferramentas para comparação de gramáticas independentes de contexto. Ou seja, calculamos $S_1(\mu_{a_1}, \ldots, \mu_{a_n})$ e $S_2(\mu_{a_1}, \ldots, \mu_{a_n})$ para um número suficientemente grande de substituições matriciais e se para todas as substituições a igualdade $S_1(\mu_{a_1}, \ldots, \mu_{a_n}) = S_2(\mu_{a_1}, \ldots, \mu_{a_n})$ é satisfeita, então concluímos que as gramáticas são equivalentes. Nesta tese não discutimos o número de substituições necessárias para concluir que as gramáticas são equivalentes com alguma probabilidade. Este assunto será objeto de futura investigação. Gostaríamos de observar que, de acordo com nossa experiência, uma substituição de matrizes aleatórias (2×2) ou (3×3) é suficiente para distinguir entre duas gramáticas independentes de contexto diferentes.

Observe também que a ideia de usar matrizes para estudar séries de potências formais não é nova (cf. [37]), mas a abordagem apresentada em [37] é bastante diferente da nossa.

O capítulo está organizado da seguinte forma. Na secção 2, demonstramos teoremas de distinguibilidade. A secção 3 contém a análise de convergência do método iterativo para equações não-lineares matriciais. As limitações do método são discutidas na secção 4. A secção 5 contém uma breve conclusão.

4.1 Teoremas de distinguibilidade

Qualquer linguagem independente de contexto pode ser definida em termos de uma série de potências formal com variáveis associativas, mas não co-

mutativas [35, 36]. Seja V_T o alfabeto terminal, $W(V_T)$ o conjunto de palavras sobre V_T, e Z_+ o conjunto de inteiros não-negativos. Uma aplicação $\phi : W(V_T) \to Z_+$ define uma série de potências formal

$$S = \sum_{P \in W(V_T)} \phi(P)P. \tag{4.4}$$

Seja μ uma aplicação de V_T para o conjunto $R^{\mathcal{N}\times\mathcal{N}}$ de matrizes $\mathcal{N} \times \mathcal{N}$. Por $P(\mu)$ vamos denotar a matriz obtida substituindo as letras $a_i \in P$ pelas matrizes μ_i e calculando o respetivo produto matricial, isto é, se $P = a_{i_1}, \ldots, a_{i_n}$, então $P(\mu) = \mu_{i_1} \ldots \mu_{i_n}$. Se a série

$$S(\mu) = \sum_{P \in W(V_T)} \phi(P)P(\mu) \tag{4.5}$$

converge, sua soma é uma matriz $\mathcal{N} \times \mathcal{N}$. Os seguintes teoremas de *distinguibilidade* formam uma base teórica para algoritmos de verificação.

4.1.1 Teorema geral de distinguibilidade

Teorema 2 (Distinguibilidade I)**.** *Seja S_1 e S_2 duas séries formais diferentes correspondentes a gramáticas independentes de contexto. Então existe um inteiro positivo $\mathcal{N}$ e uma substituição matricial $\mu : V_T \to R^{\mathcal{N}\times\mathcal{N}}$ tal que $S_1(\mu) \neq S_2(\mu)$.*

Para demonstrar o teorema precisamos de dois lemas auxiliares.

Lema 3. *Sejam U e V dois conjuntos finitos diferentes de palavras de comprimento N. Então existem um número natural $\mathcal{N}$ e uma substituição matricial $\mu : V_T \to R^{\mathcal{N}\times\mathcal{N}}$ tais que $U(\mu) \neq V(\mu)$.*

Demonstração. Seja $a_{j_1} \ldots a_{j_k} a_{i_1} \ldots a_{i_l} \in U \cup V$, onde $l \leq N$, uma palavra. Dizemos que $a_{i_1} \ldots a_{i_l}$ é uma *sub-palavra*. Designemos o conjunto de subpalavras por $\mathcal{S}$. Consideremos o conjunto de vetores ortogonais unitários $\{e_0\} \cup \{e_{i_1 \ldots i_l} \mid a_{i_1} \ldots a_{i_l} \in \mathcal{S}\}$ no espaço de dimensão adequada $\mathcal{N}$. Definimos operadores lineares μ_i, $i = \overline{1, I}$ por:

$$\mu_i e_0 = \begin{cases} e_i, & a_i \in \mathcal{S} \\ 0, & \text{caso contrário} \end{cases}$$

$$\mu_i e_{i_1 \ldots i_l} = \begin{cases} e_{i i_1 \ldots i_l}, & a_i a_{i_1} \ldots a_{i_l} \in \mathcal{S} \\ 0, & \text{caso contrário} \end{cases}$$

Seja $a_{i_1} \dots a_{i_N} \in U \cup V$ uma palavra. O operador linear correspondente tem a forma $\mu_{i_1} \dots \mu_{i_N} \in R^{\mathcal{N}\times\mathcal{N}}$. Obviamente, temos $\mu_{i_1} \dots \mu_{i_N} e_0 = e_{i_1 \dots i_N}$. desta forma,

$$U(\mu)e_0 = \sum_{(a_{i_1}\dots a_{i_N})\in U} e_{i_1\dots i_N} \neq \sum_{(a_{i_1}\dots a_{i_N})\in V} e_{i_1\dots i_N} = V(\mu)e_0,$$

Pois, os conjuntos de vetores ortogonais nas duas somas são diferentes.□

Lema 4. *O número de palavras de comprimento N gerado por uma gramática independente de contexto não excede Cq^N, onde as constantes C e q dependem da gramática.*

Demonstração. Sem perda de generalidade, a gramática tem forma normal de Greibach. Em seguida, a gramática tem M produções da forma $A \to aA_{i1} \dots A_{i_l}$, onde $A, A_{i_1}, \dots, A_{i_l} \in V_N$, $a \in V_T$ e $l \leq L$. Sem perda de generalidade $ML > 1$. A aplicação de todas as sequências possíveis de N produções gera não mais que

$$ML + (ML)^2 + \dots + (ML)^N = ML\frac{(ML)^N - 1}{ML - 1} < \frac{ML}{ML-1}(ML)^N$$

palavras. Este conjunto de palavras contém todas as palavras de comprimento N. De facto, qualquer sequência de $N + 1$ produções contém pelo menos $N + 1$ símbolos terminais. □

Demonstração do Teorema 2. Como as séries são diferentes, admitem a seguinte representação:

$$S_1 = S_0 + U + R_1 \text{ e } S_2 = S_0 + V + R_2,$$

Onde S_0 é a parte composta por palavras coincidentes de comprimento menor ou igual a N, U e V são partes diferentes compostas por palavras com comprimento igual a N, R_1 e R_2, contem termos com palavras de comprimento maior do que N.

Pelo Lema 3 existe um número natural $\mathcal{N}$ e uma substituição matricial $\mu : V_T \to R^{\mathcal{N}\times\mathcal{N}}$ tal que $U(\mu) \neq V(\mu)$. Seja $t > 0$. Nós temos

$$\Delta(t) = S_1(t\mu) - S_2(t\mu) = t^N(U(\mu) - V(\mu)) + (R_1(t\mu) - R_2(t\mu)). \quad (4.6)$$

As normas das matrizes μ_i, $i = \overline{1, I}$, construídas no Lema 3, não excedem $\sigma > 0$. Pelo Lema 4 obtemos

$$\|R_1(t\mu)\| \leq C_1(q_1\sigma t)^{N+1} \quad \text{e} \quad \|R_2(t\mu)\| \leq C_2(q_2\sigma t)^{N+1}.$$

Seja $C = \max\{C_1, C_2\}$, $q = \max\{q_1, q_2\}$ e $t < 1/(q\sigma)$. Então temos

$$\|R_1(t\mu) - R_2(t\mu)\| \le 2C\left((q\sigma t)^{N+1} + (q\sigma t)^{N+2} + \ldots\right) = 2C(q\sigma)^{N+1}\frac{t^{N+1}}{1-q\sigma t}.$$

Deste e de (4.6) vemos que $\Delta(t) \neq 0$ sempre que $t > 0$ é suficientemente pequeno. □

Em muitas situações basta considerar as substituições matriciais $\mu : V_T \to R^{2\times 2}$. Nós associamos com os símbolos $a_i \in V_T$, $i = \overline{1, I}$, pares de variáveis u_i e v_i, $i = \overline{1, I}$. Sejam

$$U = \bigcup_{\{(k'^m_N,\ldots,k'^m_1)|m=\overline{1,M}\}} a_{k'^m_N}\ldots a_{k'^m_1} \quad \text{e} \quad V = \bigcup_{\{(k''^m_N,\ldots,k''^m_1)|m=\overline{1,M}\}} a_{k''^m_N}\ldots a_{k''^m_1}$$

dois conjuntos de palavras. Consideremos dois conjuntos de polinómios associados

$$\mathcal{P}_U = \left\{ \prod_{j=l+1}^{N} u_{k'^m_j} v_{k'^m_l} \mid l = \overline{1, N} \right\}, \quad \text{e} \quad \mathcal{P}_V = \left\{ \prod_{j=l+1}^{N} u_{k''^m_j} v_{k''^m_l} \mid l = \overline{1, N} \right\}.$$

(Aqui $u_{k'^m_{N+1}} = u_{k''^m_{N+1}} = 1$.)

Nós dizemos que U e V satisfazem a condição $(\mathcal{P})$ se $\mathcal{P}_U \neq \mathcal{P}_V$.

Lema 5. *Assumindo que U e V satisfazem a condição $(\mathcal{P})$, então existe uma substituição matricial $\mu : V_T \to R^{2\times 2}$ tal que $U(\mu) \neq V(\mu)$.*

Demonstração. Consideremos as matrizes

$$\mu_i = \begin{pmatrix} u_i & v_i \\ 0 & 1 \end{pmatrix}.$$

Por indução obtemos facilmente

$$\prod_{i=1}^{N} \mu_{k_i} = \begin{pmatrix} \prod_{i=1}^{N} u_{k_i} & \sum_{i=1}^{N} \prod_{j=i+1}^{N} u_{k_j} v_{k_i} \\ 0 & 1 \end{pmatrix}.$$

Desta representação vemos que $\mathcal{P}_U \neq \mathcal{P}_V$ implica $U(\mu) \neq V(\mu)$. □

Teorema 6 (Distinguibilidade II)**.** *Seja S_1 e S_2 séries formais correspondentes a gramáticas independentes de contexto. Suponhamos que as séries admitem a seguinte representação:*

$$S_1 = S_0 + U + R_1 \quad \text{e} \quad S_2 = S_0 + V + R_2,$$

Onde S_0 é a parte de palavras coincidentes de comprimento menor ou igual a N, U e V são partes diferentes compostas por palavras com comprimento igual a N, e R_1 e R_2 contêm termos com palavras de comprimento maior do que N. Se U e V satisfazem a condição $\mathcal{P}$, então existe uma substituição matricial $\mu : V_T \to R^{2\times 2}$ tal que $S_1(\mu) \neq S_2(\mu)$.

Demonstração. Usando o Lema 5 e seguindo a demonstração do Teorema 2 obtemos o resultado. □

4.1.2 Exemplos

Sejam $U = \{aab, bab\}$ e $V = \{aba, bba\}$. Neste caso a condição $\mathcal{P}$ é satisfeita porque $u_1u_1v_2 \in \mathcal{P}_U$ e $u_1u_1v_2 \notin \mathcal{P}_V$. Por outro lado existem conjuntos de palavras que não é possível distinguir com ajuda de matrizes 2×2. Por exemplo, para qualquer escolha de matrizes 2×2, μ_1 e μ_2 tem-se

$$\mu_1\mu_1\mu_2\mu_2\mu_1 + \mu_1\mu_2\mu_1\mu_1\mu_2 + \mu_2\mu_1\mu_2\mu_1\mu_1$$

$$= \mu_1\mu_1\mu_2\mu_1\mu_2 + \mu_1\mu_2\mu_2\mu_1\mu_1 + \mu_2\mu_1\mu_1\mu_2\mu_1.$$

(É fácil verificar esta igualdade usando, por exemplo, o sistema de cálculo analítico Maxima.) Portanto, as linguagens

$$\{aabba, abaab, babaa\} \text{ and } \{aabab, abbaa, baaba\} \tag{4.7}$$

não é possível distinguir usando matrizes 2×2. No entanto, substituindo matrizes aleatórias 3×3 é fácil mostrar que as linguagens são diferentes (4.7).

4.1.3 Testes para análise de linguagens finitas sobre alfabeto $\{a, b, c\}$

A fim de caracterizar a abrangência do método de comparação de gramáticas usando a substituição por matrizes 2×2 e 3×3, fizemos um conjunto de experiências cujo primeiro objectivo era o de encontrar exemplos de falsos positivos (gramáticas não equivalentes cujas linguagens não são distinguíveis por substituição de matrizes 2×2).

As experiências realizadas envolveram:

- a geração de pares de gramáticas simples (usando um alfabeto pequeno, um número pequeno de produções, sem usar não-terminais do lado direito)
- comparação das séries formais (finitas) após a substituição dos terminais por matrizes 2×2 ou 3×3

A escolha deste tipo de gramáticas (correspondentes a linguagens finitas) deve-se a que mesmo com um pequeno alfabeto é fácil de ver que o número de gramáticas que podemos gerar é demasiado grande para um estudo exaustivo.

Para ajudar à realização de uma série de experiências de comparação, construiu-se uma ferramenta (**grampermgen** – grammar permutations generator) para gerar gramáticas a partir de um conjunto de restrições sobre o seu alfabeto, número de produções e comprimentos de cada produção. Exemplo de uso:

```
grampermgen ALF=aabb  COMP=2-4   PAR=2-3
```

indicando que:

- o lado direito de cada produção tem entre 2 e 4 elementos,
- o número de produções pode variar entre 2 e 3,
- o alfabeto é $\{a, b\}$,
- em cada lado direito de produção, o número máximo de ocorrências de a é 2 e de b também é 2.

Este comando gera 80 gramáticas diferentes, por exemplo:

$$\begin{array}{rcl} S & \rightarrow & abab \\ & | & abba \\ & | & bbaa \\ & ; & \end{array}$$

O gerador remove automaticamente as gramáticas (equivalentes) que apenas diferem na ordem das produções.

O processo natural de comparação seria comparar cada gramática com todas as outras. No caso do exemplo, isto corresponde a $80 \times 79/2 = 3160$ comparações. No entanto constatou-se que mesmo com um número pequeno de comparações o tempo necessário atinge valores inviáveis. Para o caso

referido, o tempo demorado corresponde a cerca de 308s, ou seja cerca de 5 minutos.

Seguidamente optamos por comparar apenas as gramáticas que tivessem igual número de produções e comprimento de palavras – ou seja passamos a dividir as gramáticas por classes com idênticos números, e a realizar comparações apenas dentro da mesma classe.

As gramáticas de diferentes classe são facilmente distinguíveis mesmo substituindo cada terminal por um número aleatório tornando desnecessário o teste com matrizes. (Obviamente a distinguibilidade por matrizes $\mathcal{N} \times \mathcal{N}$ implica a distinguibilidade por matrizes $(\mathcal{N} + \infty) \times (\mathcal{N} + \infty)$.)

No exemplo apresentado aparecem 6 classes, obtendo-se uma significativa redução do número comparações (ver tabela abaixo).

Classe	número de gram.	número de testes
2-2	6	15
2-3	15	105
2-4	15	105
3-2	4	6
3-3	20	190
3-4	20	190
total		611

Mesmo dentro de uma mesma classe, há situações em que a comparação por matrizes se torna desnecessária:

$$\begin{array}{rcl} G1 & \rightarrow & aa \\ & | & bb \\ & ; & \\ G2 & \rightarrow & aa \\ & | & ab \\ & ; & \end{array}$$

Estas duas gramáticas pertencentes à classe 2-2 são obviamente diferentes já que mesmo no domínio dos reais $a^2 + b^2 \neq a^2 + ab$.

Esta constatação levou-nos à criação de subclasses caracterizadas pelo número de ocorrências de cada terminal (multiset dos terminais).

Subclasse	número de gram.	número de testes
2-2 a:1 b:3	2	1
2-2 a:2 b:2	2	1
2-2 a:3 b:1	2	1
2-3 a:2 b:4	3	3
2-3 a:3 b:3	9	36
2-3 a:4 b:2	3	3
2-4 a:4 b:4	15	105
3-2 a:2 b:4	1	0
3-2 a:3 b:3	2	1
3-2 a:4 b:2	1	0
3-3 a:3 b:6	1	0
3-3 a:4 b:5	9	36
3-3 a:5 b:4	9	36
3-3 a:6 b:3	1	0
3-4 a:6 b:6	20	190
total		431

Embora a redução obtida para este exemplo seja apenas de 30%, para gramáticas de maior dimensão, esta redução torna-se mais significativa.

Durante o processo de procura de pares de gramáticas cuja comparação desse falso positivo (pares problemáticos), constatou-se que o número de comparações pode ser resumido atendendo a que:

- se um par de gramáticas $P = (G_1, G_2)$ é problemático, o par obtido por adição de uma mesma produção a cada uma das gramática dá também um par problemático;
- se um par de gramáticas $P = (G_1, G_2)$ é problemático, o par obtido por adição a cada produção de um mesmo prefixo ou sufixo, dá também um par problemático;
- se um par de gramáticas $P = (G_1, G_2)$ é problemático, o par obtido por troca dos elementos do alfabeto (exemplo: substituição de $a \to b; b \to a$), dá também um par problemático.

Estas propriedades foram usadas para diminuir o número de comparações e as duas primeiras foram usadas para simplificar as gramáticas dos pares problemáticos.

Usando esta ferramenta geradora de gramáticas e efectuando as referidas optimizações de escolha dos pares a comparar, foram realizadas várias experiências, que a seguir resumimos.

- **Alfabeto=aabb comp=2..4 par=2..3**

  Não foram encontrados falsos positivos em 2×2.

- **Alfabeto=aaabbb comp=5 par=5**

  Esta experiência encontrou os primeiros falsos positivos com matrizes 2×2.

- **Alfabeto=aaabb comp=5 par=5**

  Esta experiência dá os mesmos pares problemáticos relevantes que a experiência anterior, mas efectuando muito menos comparações.

- **Alfabeto=aaabbc comp=6 par=3**

  Esta experiência encontrou diversos pares problemáticos que após as simplificações referidas anteriormente, permitiu calcular os pares problemáticos relevantes para comprimentos ≤ 6

  Realizamos cerca de $47 \cdot 10^6$ testes para analisar as linguagens finitas sobre o referido alfabeto contendo no máximo três palavras de comprimento menor ou igual a seis. Dentro do subconjunto de linguagens de comprimento menor ou igual a 5, somente as seguintes linguagens não podem ser distinguidos usando substituições matrizes 2×2. Nomeadamente, o par das linguagens

  $$S_1 = \{aabca, abaac, bacaa\} \quad \text{e} \quad S_2 = \{aabac, abcaa, baaca\} \tag{4.8}$$

  e outros pares obtidos como resultado da permutação das letras $\{a, b, c\}$ e/ou substituição de c por a ou b. Denotaremos este conjunto de pares de linguagens por L

  Todos os pares problemáticos encontrados nesta experiência, (tanto este par problemático como todos os que têm comprimento 6), foram distinguidos por matrizes 3×3.

- **Alfabeto=aaabbc comp=7 par=3**

  Esta experiência pretendeu procurar pares problemáticos no domínio de matrizes 3×3. O número de pares a comparar é tão elevado que esta experiência não vai ficar concluída no espaço temporal desta tese. Portanto optamos por uma abordagem de comparação por amostragem: construímos uma sequência aleatória de pares.

  Até ao momento não foram encontrados pares problemáticos no domínio de matrizes 3×3.

A quantidade de comparações envolvidas e o respectivo esforço computacional, obrigou à utilização de um cluster no qual foram lançados alguns milhares de *jobs* com durações médias de cerca de 20 horas.

Os testes realizados até ao momento correspondem a um tempo de cálculo de cerca de 2 meses.

Neste conjunto de experiências usamos um alfabeto $\{a, b, c\}$ para procura de pares de gramáticas problemáticos. Na realidade a mesma situação ocorre quando em vez de a, b, c temos um qualquer símbolo não terminal e portanto estes pares problemáticos descrevem os padrões que são problemáticos em gramáticas.

Por exemplo, o par de linguagens (4.8) pode ser usado para construir infinitas linguagens que não podem ser distinguidas usando matrizes 2×2. Um desses exemplos é o par

$$\begin{array}{rcl} S & \rightarrow & aabAa \\ & | & abaaA \\ & | & baAaa \\ & ; & \\ A & \rightarrow & aA \\ & | & b \\ & ; & \end{array}$$

e

$$\begin{array}{rcl} S & \rightarrow & aabaA \\ & | & abAaa \\ & | & baaAa \\ & ; & \\ A & \rightarrow & aA \\ & | & b \\ & ; & \end{array}$$

Olhando agora para gramáticas reais, notemos que as situações problemáticas ocorrerão apenas quando direta ou indiretamente forem usados padrões análogos aos referidos.

Observe também que em todas as gramáticas de várias linguagens de programação e de exercícios educacionais que analisamos, a distinção sempre foi possível usando matrizes (2×2).

4.1.4 Teoremas de distinguibilidade baseados na comparação de palavras curtas

Em muitas situações a diferença entre duas gramáticas pode ser detectada comparando palavras curtas utilizando substituição de matrizes 2×2.

Teorema 7 (Distinguibilidade III)**.** *Sejam S_1 e S_2 séries formais correspondentes a gramática independente de contexto sobre o alfabeto terminal $\{a, b, c\}$ e $N \leq 5$. Suponha-se que as séries admitem a seguinte representação:*

$$S_1 = S_0 + U + R_1 \quad \text{e} \quad S_2 = S_0 + V + R_2,$$

Onde S_0 é a parte de palavras coincidentes de comprimento menor ou igual a N, U e V são partes diferentes compostas por não mais de três palavras com comprimento igual a N, R_1 e R_2 contêm termos com palavras de comprimento maior do que N. Se o par U e V não coincidir com um dos pares de L, então existe uma substituição matricial $\mu : V_T \to R^{2\times 2}$ tal que $S_1(\mu) \neq S_2(\mu)$.

Demonstração. Seguindo a demonstração do Teorema 2 obtemos o resultado. □

4.2 Sistema de equações não-lineares matriciais

Sabe-se que a qualquer gramática independente de contexto corresponde um sistema de equações não-lineares [35, 36] que permite obter as respectivas séries de potências formais através de iterações sucessivas. Os termos da série são as palavras da respectiva linguagem. Esta correspondência entre séries e sistemas de equações não-lineares torna possível calcular efetivamente as somas das séries para uma substituição matricial $\mathcal{N} \times \mathcal{N}$. Sejam X_i, $i = \overline{0, m}$, não-terminais de uma gramática independente de contexto e sejam P_j^i, $i = \overline{0, m}$, $j = \overline{1, l_i}$, as palavras que aparecem no lado direito das produções com

os lados esquerdos X_i. O sistema de equações correspondente à gramática tem a forma

$$\begin{aligned} X_1 &= P_1^1 + \ldots + P_{l_1}^1, \\ &\vdots \\ X_m &= P_1^m + \ldots + P_{l_m}^m. \end{aligned} \tag{4.9}$$

Substituindo os símbolos do alfabeto terminal $a_k \in P_j^i$ por matrizes μ_k, obtemos um sistema de equações não-lineares matriciais. Este sistema, $X = F(X)$, onde $X = (X_1, \ldots, X_m)$, pode ser resolvido usado um procedimento iterativo: $X^{k+1} = F(X^k)$, $X^0 = 0$, ou usando o método de Newton. Sendo este último mais rápido. Como vamos ver posteriormente a convergência das iterações pode ser garantida para gramáticas na forma normal reduzida de Chomsky e forma normal de Greibach. Notemos que no caso das gramáticas regulares, o sistema (4.9) é linear.

4.2.1 Convergência do método iterativo

Consideremos uma gramática independente de contexto com produções

$$\begin{aligned} X_i &\to \mathcal{P}_j^{0,i}, \;\; i = \overline{1,n}, \; j = \overline{1, J_i^{\mathcal{P}}} \\ X_i &\to p_j^i, \;\; i = \overline{1,n}, \; j = \overline{1, J_i^{p}}, \end{aligned}$$

Onde $\mathcal{P}_j^{0,i} \in W(V_N \cup V_T) \setminus W(V_T)$ e $p_j^i \in W(V_T)$, $p_j^i \neq \varepsilon$. Suponha-se que as palavras $\mathcal{P}_j^{0,i}$ contêm mais do que uma letra. (Por exemplo, as gramáticas têm forma normal reduzida de Chomsky ou de Greibach.). A estrutura destas palavras pode ser descrita de seguinte maneira:

$$\mathcal{P}_j^{l,i} = q_j^{l+1,i} X_{k_j^{l+1,i}} \mathcal{P}_j^{l+1,i}, \quad l = \overline{0, L_j^i},$$

Onde $\mathcal{P}_j^{l,i} \in W(V_N \cup V_T) \cup \{\emptyset\}$ e $q_j^{l,i} \in W(V_T) \cup \{\emptyset\}$. O sistema de equações formais correspondente tem a estrutura seguinte:

$$X_i = F_i(X_1, X_2, \ldots, X_n) = \sum_j \mathcal{P}_j^{0,i} + \sum_j p_j^i \tag{4.10}$$

Para simplificar as notações denotamos por P a matriz $P(\mu)$ obtida substituindo os símbolos a_i e $X_i \in P$ pelas matrizes μ_i e ξ_i, respectivamente. Usamos a notação $\tilde{P}$ quando os símbolos $X_i \in P$ são substituídos pelas matrizes $\tilde{\xi}_i$. Sejam $X = (X_1, \ldots, X_n)$ e $\tilde{X} = (\tilde{X}_1, \ldots, \tilde{X}_n)$ duas coleções de n matrizes $\mathcal{N} \times \mathcal{N}$. Então, usando nossas notações, temos

$$\mathcal{P}_j^{0,i} - \tilde{\mathcal{P}}_j^{0,i} = q_j^{1,i}(X_{k_j^{1,i}} \mathcal{P}_j^{1,i} - \tilde{X}_{k_j^{1,i}} \tilde{\mathcal{P}}_j^{1,i}) = q_j^{1,i}(X_{k_j^{1,i}} - \tilde{X}_{k_j^{1,i}}) \mathcal{P}_j^{1,i} + \tilde{X}_{k_j^{1,i}} (\mathcal{P}_j^{1,i} - \tilde{\mathcal{P}}_j^{1,i})$$

$$= q_j^{1,i}(X_{k_j^{1,i}} - \tilde{X}_{k_j^{1,i}})\mathcal{P}_j^{1,i} + \tilde{X}_{k_j^{1,i}}(q_j^{2,i}(X_{k_j^{2,i}} - \tilde{X}_{k_j^{2,i}})\mathcal{P}_j^{2,i} + \tilde{X}_{k_j^{2,i}}(\mathcal{P}_j^{2,i} - \tilde{\mathcal{P}}_j^{2,i}))$$
$$= \ldots = Y_j^{1,i}(X_{k_j^{1,i}} - \tilde{X}_{k_j^{1,i}})Z_j^{1,i} + \ldots + Y_j^{n,i}(X_{k_j^{n_j^i,i}} - \tilde{X}_{k_j^{n_j^i,i}})Z_j^{n,i}, \quad (4.11)$$

Onde $Y_j^{l,i}, Z_j^{l,i} \in W(V_N \cup V_T)$ e n_j^i é o número de não-terminais na palavra $\mathcal{P}_j^{0,i}$. Suponhamos que as normas de todas as matrizes a_i, X_i, e $\tilde{X}_i$ não excedem $\delta > 0$ e que $\bar{n}\delta < 1$, onde $\bar{n} = \sum_{i,j} n_j^i$. Então da representação (4.11) obtemos

$$\max_{i=\overline{1,n}} \|F_i(X) - F_i(\tilde{X})\| \leq \bar{n}\delta \max_{j=\overline{1,n}} \|X_j - \tilde{X}_j\| \quad (4.12)$$

Seja $\tilde{X} = 0$. Então se $\|X_i\| < \delta$, temos

$$\max_{i=\overline{1,n}} \|F_i(X)\| \leq \bar{n}\delta \max_{j=\overline{1,n}} \|X_j\| < \bar{n}\delta^2 < \delta.$$

Consideremos o conjunto fechado $\mathcal{B} = \{X \mid \max_{j=\overline{1,n}} \|X_j\| \leq \delta\}$ no espaço de matrizes $X = (X_1, \ldots, X_n)$. Acabamos de demonstrar que $F(\mathcal{B}) \subset \mathcal{B}$ e que F é uma aplicação contractante, e portanto, tem um único ponto fixo $\hat{X} = F(\hat{X}) \in \mathcal{B}$. Este ponto fixo é limite da sucessão

$$X^{k+1} = F(X^k), \quad k = 0, 1, \ldots, \quad X^0 = 0. \quad (4.13)$$

Acabamos de demonstrar o teorema seguinte.

Teorema 8 (De convergência)**.** *Suponhamos que o sistema de equações não-lineares que corresponde a uma gramática independente de contexto tem forma (4.10) e que as palavras com símbolos não-terminais, $\mathcal{P}_j^i$, contêm mais de um símbolo. Então, substituindo os símbolos terminais por matrizes com uma norma suficientemente pequena (menor que δ), podemos garantir a convergência da sucessão (4.13) para uma solução única do sistema de equações matriciais (4.10).*

Exemplo

Consideremos a gramática

$$\begin{array}{rcl} S & \rightarrow & SaA \\ & | & a \\ & ; & \\ A & \rightarrow & cSd \\ & | & b \\ & ; & \end{array}$$

O respectivo sistema de equações é

$$\begin{array}{l} S = F_S = SaA + a, \\ A = F_A = cSd + b. \end{array} \tag{4.14}$$

As condições do Teorema 8 estão satisfeitas (as palavras AaS e cSd têm mais do que uma letra). Portanto, o sistema pode ser resolvido usando o método iterativo sempre que os símbolos a, b, c e d forem substituídos por matrizes com uma norma suficientemente pequena.

4.2.2 Outros casos de aplicação do método iterativo

Em muitos casos a gramática pode ter produções da forma $A \to B$. O método iterativo pode ser aplicado também ao sistema correspondente de equações após alguma transformação. Ou seja, suponhamos que o sistema tem a forma

$$X = F(X) + \Lambda X, \tag{4.15}$$

onde F tem a forma (4.10) considerada anteriormente e $\Lambda : R^{(\mathcal{N}\times\mathcal{N})n} \to R^{(\mathcal{N}\times\mathcal{N})n}$ é uma aplicação linear tal que exista a aplicação inversa $(I - \Lambda)^{-1}$. Então o sistema (4.15) é equivalente ao sistema

$$X = (I - \Lambda)^{-1} F(X).$$

Obviamente, a aplicação $X \to (I - \Lambda)^{-1} F(X)$ é contactante e transforma $\mathcal{B} = \{X \mid \max_{j=\overline{1,n}} \|X_j\| \leq \delta\}$ em $\mathcal{B}$, sempre que os símbolos terminais forem substituídos por matrizes com normas suficientemente pequenas.

Exemplo

Consideremos a gramática

$$\begin{array}{rcl} S & \to & SaA \\ & | & A \\ & ; & \\ A & \to & cSd \\ & | & b \\ & ; & \end{array}$$

que é uma solução correta de um exercício. O sistema de equações correspondente é:

$$\begin{aligned} S &= SaA + A, \\ A &= cSd + b. \end{aligned} \tag{4.16}$$

Abaixo apresentamos três outras respostas possíveis.

Solução correta A gramática a seguir é diferente, mas gera a mesma linguagem:

$$\begin{array}{rcl} S & \rightarrow & AaS \\ & | & A \\ & ; & \\ A & \rightarrow & cSd \\ & | & b \\ & ; & \end{array}$$

O sistema de equações correspondente é

$$\begin{aligned} S &= AaS + A, \\ A &= cSd + b. \end{aligned} \tag{4.17}$$

Resposta errada A seguinte gramática não gera a mesma linguagem (não gera a palavra *cbabd*):

$$\begin{array}{rcl} S & \rightarrow & SaA \\ & | & A \\ & ; & \\ A & \rightarrow & cAd \\ & | & b \\ & ; & \end{array}$$

O sistema de equações correspondente é

$$\begin{aligned} S &= SaA + A, \\ A &= cAd + b. \end{aligned} \tag{4.18}$$

Gramática ambígua A seguinte gramática gera a mesma linguagem, mas é ambígua (a palavra *baba* pode ser gerada de diferentes maneiras):

$$\begin{array}{rcl} S & \to & SaS \\ & | & A \\ & ; & \\ A & \to & cSd \\ & | & b \\ & ; & \end{array}$$

O sistema correspondente tem a forma

$$\begin{array}{l} S = SaS + A, \\ A = cSd + b. \end{array} \tag{4.19}$$

O sistema (4.16) é equivalente ao sistema

$$\begin{pmatrix} S \\ A \end{pmatrix} = \begin{pmatrix} I & I \\ 0 & I \end{pmatrix}^{-1} \begin{pmatrix} SaA \\ cSd + b \end{pmatrix}.$$

(Aqui I é a matriz identidade.) Substituindo os símbolos a, b, c e d por matrizes de norma suficientemente pequena obtemos um sistema resolúvel pelo método iterativo. Analogamente podemos transformar e resolver os outros sistemas. Começando o processo iterativo com $S = A = 0$ e resolvendo sistemas (4.16), (4.17), (4.18) e (4.19) vemos que a diferença entre S componentes de solução de sistemas (4.16) e (4.17) é zero, enquanto para o par (4.16) e (4.18) ou (4.16) e (4.19) a diferença não é zero. Isso permite distinguir claramente entre respostas certas e erradas.

Observe que o intervalo onde os componentes das matrizes são gerados deve ser (a) suficientemente pequeno para garantir a convergência do processo iterativo, (b) grande o suficiente para distinguir entre duas linguagens diferentes.

Outro caso importante trata das gramáticas com produções da forma $A \to \varepsilon$. Neste caso, algumas equações têm a forma

$$X_i = F_i(X_1, X_2, \ldots, X_n) + I$$

Introduzindo novas variáveis $Y_i = X_i - I$ em muitas situações é possível transformar o sistema em um sistema que satisfaça as condições do Teorema 8.

Exemplo

Consideremos a gramática

$$\begin{array}{rcl} S & \rightarrow & SaA \\ & | & b \\ & ; & \\ A & \rightarrow & cSd \\ & | & \varepsilon \\ & ; & \end{array}$$

O sistema correspondente de equações não-lineares é

$$\begin{aligned} S &= SaA + b \\ A &= cSd + I \end{aligned}$$

Introduzindo a nova variável $B = A - I$ obtemos o sistema

$$\begin{aligned} S &= SaB + Sa + b \\ B &= cSd \end{aligned}$$

Que pode ser resolvido pelo método iterativo.

Se as transformações descritas anteriormente não levarem a um sistema abrangido pelo Teorema 8, sempre é possível usar a transformação à forma normal reduzida de Chomsky ou à forma normal de Greibach.

4.3 Limitações do método iterativo

Em alguns casos, o método iterativo não pode distinguir entre duas gramáticas diferentes. Isso acontece quando consideramos gramáticas com palavras muito longas de $W(V_T)$. O ponto é que a precisão do computador não é suficiente para calcular corretamente produtos de muitos números pequenos. Consideremos a seguinte gramática

$$
\begin{array}{rcl}
S & \rightarrow & AS \\
 & | & BS \\
 & | & B \\
 & ; & \\
A & \rightarrow & a_1 a_2 \ldots a_n \\
 & ; & \\
B & \rightarrow & a_1 \\
 & | & a_2 \\
 & | & \ldots \\
 & | & a_n \\
 & ; &
\end{array}
$$

O sistema correspondente de equações matriciais é

$$
\begin{array}{l}
S = AS + BS + B, \\
A = a_1 a_2 \ldots a_n, \\
B = a_1 + a_2 \ldots + a_n.
\end{array}
\tag{4.20}
$$

A segunda gramática é

$$
\begin{array}{rcl}
S & \rightarrow & AS \\
 & | & BS \\
 & | & B \\
 & ; & \\
A & \rightarrow & a_2 a_1 \ldots a_n \\
 & ; & \\
B & \rightarrow & a_1 \\
 & | & a_2 \\
 & | & \ldots \\
 & | & a_n \\
 & ; &
\end{array}
$$

com o correspondente sistema de equações

$$\begin{aligned} S &= AS + BS + B, \\ A &= a_2 a_1 \ldots a_n, \\ B &= a_1 + a_2 \ldots + a_n. \end{aligned} \tag{4.21}$$

Os sistemas (4.20) e (4.21) são equivalentes às equações

$$S = a_1 a_2 \ldots a_n S + (a_1 + a_2 + \ldots + a_n)S + (a_1 + a_2 + \ldots + a_n)$$

e

$$S = a_2 a_1 \ldots a_n S + (a_1 + a_2 + \ldots + a_n)S + (a_1 + a_2 + \ldots + a_n),$$

respectivamente. Para garantir a convergência das iterações temos que impor a restrição

$$\|a_1\| + \|a_2\| + \ldots + \|a_n\| = \alpha < 1.$$

Da desigualdade das médias aritmética e geométrica obtemos

$$\sqrt[n]{\|a_1\| \ldots \|a_n\|} \leq \frac{\|a_1\| + \ldots + \|a_n\|}{n}.$$

Portanto, para n suficientemente grande, temos

$$\|a_1\| \ldots \|a_n\| \leq \frac{\alpha^n}{n^n} < \delta,$$

Onde $\delta > 0$ é a precisão do computador. Assim, o computador interpreta os processos iterativos

$$S_{k+1} = a_1 a_2 \cdots a_n S_k + (a_1 + a_2 + \cdots + a_n)S_k + (a_1 + a_2 + \cdots + a_n)$$

e

$$S_{k+1} = a_2 a_1 \cdots a_n S_k + (a_1 + a_2 + \cdots + a_n)S_k + (a_1 + a_2 + \cdots + a_n)$$

como o mesmo processo

$$S_{k+1} = (a_1 + a_2 + \cdots + a_n)S_k + (a_1 + a_2 + \cdots + a_n)$$

E, portanto, o método não permite distinguir entre duas gramáticas.

Contudo, o método pode ser aplicado a gramáticas de tamanho real com um resultado satisfatório. Para testar esta abordagem em casos de tamanho real, usamos um gramática semelhante à linguagem C simplificada [24] com cerca de 44 símbolos não terminais e 104 produções. Foi feita a comparação desta gramática com outras versões a ela equivalentes e com outras variantes que apresentavam pequenas diferenças na linguagem coberta. Através da geração de matrizes aleatórias (2×2) com valores no intervalo $[0, 0.01]$, conseguimos em todos os casos resolver os correspondentes sistemas de equações matriciais e correctamente distinguir as situações de equivalência e de não equivalência das gramáticas.

4.4 Conclusão

Neste capítulo abordamos o problema da comparação de gramáticas independentes de contexto do ponto de vista de análise matemática. A substituição de letras terminais por matrizes permite reduzir o problema de comparação à solução numérica de sistema de equações não-lineares matriciais. Além da elegância do processo, este método constitui uma base sólida para a construção de algoritmos e ferramentas para a comparação de gramáticas independentes de contexto. As nossas experiências com um protótipo construído, mostram que o uso deste método de comparação com matrizes (2×2) e (3×3) em problemas que aparecem em *e-learning* e até mesmo em casos de gramáticas grandes é muito eficiente.

Capítulo 5

Sistema / ferramentas

Como falamos no capítulo 3, é muito importante ter um rico *kit* de ferramentas com funções capazes de manipular gramáticas e outros componentes ligados ao domínio das linguagem.

Neste capítulo descrevemos as ferramentas principais desenvolvidas no âmbito desta tese. As diversas funções vão ser agrupadas nas seguintes secções:

- Definição de gramáticas;
- Reconhecimento de frases;
- Geração de frases;
- Comparação de gramáticas;
- *Pretty print* de gramáticas;

Complementarmente as funções ligadas ao processamento de meta-gramáticas foram apresentadas no capítulo 3.

5.1 Definição de gramáticas

Tanto do ponto de vista das experiências realizadas como do ponto de vista da geração de exercícios, constatamos a necessidade de criar uma sintaxe simples e concisa para definição de gramáticas independentes de contexto e de criar um conjunto de funções que as reconhecesse.

Seguidamente apresenta-se alguns exemplos e sua formalização.

Considere-se a seguinte gramática, escrita em notação **Mgbeg**:

```
#tit: exemplo 1

S : S a A
  | A ;
A : c S d
  | b
  | a b c ;
```

Para uma otimização de escrita, convencionou-se que:

- sempre que possível seguimos a notação habitual das ferramentas ligadas a gramáticas independentes de contexto – ':' ou '→' (deriva em), '|' (alternativa) e ';' (fim de produção) – são símbolos especiais com o significado habitual em notação gramatical (exemplo, Bison [19], Yacc [26]);

- todo o símbolo que não aparece como lado esquerdo de uma produção é um símbolo terminal;

- o primeiro símbolo não terminal (neste caso S) é o axioma da gramática.

Por questões práticas permite-se a inclusão de comentários:

- comentários de linha (`#....até fim de linha`);

- comentários de fim de ficheiros (deste `__END__` até fim de ficheiro);

- comentários para definição de metadados associados à gramática (Exemplo: título, data, observações, etc).

Existe uma estreita conexão entre a geração formal e os dispositivos de reconhecimento de linguagem. Os mecanismos de geração formal de linguagem são comummente usados para descrever a sintaxe da linguagem. As regras de sintaxe especificam quais sequências pertencem a linguagem. Para descrever concisamente a sintaxe de uma linguagem de programação foi e continua a ser universalmente usado o método BNF (Forma de Backus-Naur) [38].

Sebesta [38] afirma que a BNF é uma metalinguagem, por ser uma linguagem que descreve outra linguagem. A BNF usa abstrações para estruturas sintáticas que nada mais é que um conjunto de regras para descrever a linguagem.

Para comodidade do leitor, resumimos a notação anterior numa tabela:

Notação	Descrição
;	separa as produções
:,→	separa *lhs* de uma produção de seus *rhss*
|	separação das várias alternativas de *rhss*
`#`	identifica comentários
`#identificador:valor`	metadados (título, data, ...)
`__END__`	tudo que estiver após é considerado comentário

De um modo mais formal um texto que defina uma gramática segue a seguinte sintaxe:

```
gram → prod | prod gram

prod → lhs ':' rhss ';'

lhs → símbolo

rhss → rhs
     | rhs '|' rhss

rhs →  ε
    | símbolo rhs
```

Alguma extensões serão apresentadas ao longo deste capítulo para suprir necessidade práticas.

Internamente uma gramática é representada por um *mapping* entre os elementos não terminais e as produções a eles associadas:

```
nt → (rhss  : [[T|NT]];     ## rhss : lista dos rhs -- lados direitos associados
       probs : [real];)     ## probs : peso associado a cada produção
```

Em complemento é ainda guardado a seguinte informação:

```
'_order' → [NT];            ## lista contendo a ordem de definição dos não terminais
'_meta' → (id → valor)      ## metadados contidos na gramática
```

necessária para algumas funções complementares.

Como resultado do reconhecimento da gramática apresentada no início desta secção, a função **gramrec** constrói a seguinte representação interna:

```
( S → (rhss → [ [S a A ],       ## lados direitos
                [ A ]],
      probs → [ 1/2, 1/2 ])     ## probabilidades de aplicação da produção
  A → (rhss → [ [ c S d],
                [ b ],
                [ a b c]],
```

```
      probs → [ 1/3, 1/3, 1/3 ])
  _order → [S,A],                ## ordem das definições
  _meta → [ tit → "exemplo 1"]   ## metadados
)
```

Para realizar o reconhecimento de gramáticas, construíram-se duas funções:

gramrec(string) → gramática — função que dada uma string contendo a descrição de uma gramática, reconhece a gramática, isto é constrói uma representação formal da gramática em computador.

gramrecfile(file) → gramática — função que constrói uma gramática a partir de um ficheiro. Após a leitura do conteúdo do ficheiro, esta função usa a função anterior.

O algoritmo de reconhecimento de gramática é um parser que seguindo a definição atrás apresentada constrói uma representação interna da gramática que vai ser usada nas diversas funções do módulo.

Seguidamente apresenta-se uma versão simplificada do algoritmo da função **gramrec**:

Algorithm 4: gramrec(s:string) – reconhece a gramática;

Input: s: string com a gramática
Output: g: gramática em representação para computador
begin
  $g['\_meta'] \leftarrow$ extrair comentário de metadata se existirem
  Remover os comentários do tipo `#....`
  Remover os comentários do tipo `__END__ ....`
  $gp \leftarrow split(s,';')$ //separa os grupos de produções
  for $g1 \in gp$ **do** //para cada grupo
    $(nt, def) \leftarrow split(g1,':')$ //separar em (não terminal, definição)
    $push(g['\_order'], nt)$ //preservar a ordem de definição dos nt
    $rhss \leftarrow split('def','|')$ //calcula lista dos rhs
    for $rhs \in rhss$ **do**
      $lista \leftarrow split(rhs, espaos)$ //lista símbolos T ∪ NT
      $push(g[nt]['prob'], getprob(lista))$ //separa a probabilidade
      $push(g[nt]['rhss'], lista)$
  return g

O algoritmo 4 não faz mais que extrair e armazenar os vários componentes através de:

- extrair a metadata (titulo, autor, data, etc) se existir;

- separar nos grupos que definem cada símbolo não terminal;
- dentro de cada grupo, separar os diversos lados direitos da produção (lhs);
- dentro de cada lado direito, separar a lista de símbolos que o constituem.

No algoritmo omitimos os detalhes referentes ao reconhecimento e geração de *símbolos com valor* (exemplo: inteiros, letras, listas enumeradas).

5.2 Reconhecimento de frases

Para conseguirmos verificar se uma frase pertence a uma linguagem gerada pela gramática G, a abordagem que está a ser usada é:

1. a partir da gramática, construir um ficheiro *parser* com a sintaxe Bison e um *analisador léxico* com a sintaxe Flex (função **bisongram**);
2. a partir destes ficheiros gera-se um programa C validador (usando as ferramentas Bison [19] e Flex [32])
3. compila-se o programa C gerando um programa validador;
4. corre-se o referido validador sobre as frases a analisar (função **validsent**).

Normalmente as ferramentas geradas pelo Bison, abortam quando surge um erro sintático. No nosso caso foi necessário fazer um cuidado controle dos erros sintáticos, léxicos, e outros de modo a tentar dar sempre uma resposta. O validador em situação de ocorrências de erro retorna "*not ok*". Caso contrário retorna "*ok*".

A ferramenta Bison [19], por omissão gera reconhecedores sintáticos Bottom-Up LALR1 [17]. A generalidade das gramáticas não ambíguas são reconhecíveis por métodos LALR1. No entanto nas experiências práticas realizadas, deparamos com problemas ligados a algumas gramáticas simples que não pertencem ao conjunto das gramáticas LALR1 – ou seja, dão conflitos LALR1 e o validador é incapaz de reconhecer corretamente algumas frases exemplo.

Consideres-se a seguinte gramática das frases que contêm tantos a como b:

$$\begin{array}{rcl} E & \rightarrow & aEbE \\ & | & bEaE \\ & | & \varepsilon \\ & ; & \end{array}$$

Esta gramática também não é reconhecível pelos reconhecedores LALR1.

Analogamente, se definirmos uma gramática para as frases capicuas contendo qualquer número de a ou b: $\{f \mid f = reverse(f)\}$, exemplo "abbcbba":

$$\begin{array}{rcl} E & \rightarrow & aEa \\ & | & bEb \\ & | & cEc \\ & | & c \\ & | & a \\ & | & b \\ & | & \varepsilon \\ & ; & \end{array}$$

também cai no conjunto não reconhecível por métodos LALR1.

Naturalmente, as gramáticas ambíguas também levantam problemas de reconhecimento LALR1.

De um modo simples os conflitos LALR correspondem à possibilidade de seguir dois (ou mais) caminhos no processo de reconhecimento sintático.

A abordagem usada foi recorrer ao método GLR (Generalized LR parser) [41, 19] capaz de lidar com gramáticas com conflitos não resolvidos, à custa de replicar o reconhecedor seguindo em paralelo as várias alternativas e fundindo os estados idênticos quando tal for possível.

Em síntese, a validação de frases envolve as seguintes funções:

bisongram: gramática → bison-parser — gera e compila um analisador Bison [19] simples para a gramática;

validsent: gramática, frase → boolean — verifica se a frase é válida, usando o reconhecedor anterior.

A função bisongram é composta por duas subfunções:

bisonflex: gramática → fbison, fflex — gera reconhecedores Bison e Flex a partir da gramática de entrada;

compila: fbison, fflex — cria um programa validador a partir dos arquivos Bison e Flex anteriores.

Segue a apresentação do algoritmo da função **bisonflex**:

Algorithm 5: bisonflex(g,F,L) – reconhecedor sintático tipo GLR(1);

```
Input: g: gramática
Input: F, L : nome dos ficheiros onde bison / flex são escritos
Output: ficheiros F, L
begin
    flex ←' '                          //conteúdo flex
    bison ←' '                         //conteúdo bison
    tokens ← ()                          //símbolos T
    specials ← ()                  //caracteres especiais flex
    ... adicionar os símbolos T predefinidos
    for nt ∈ g._order do                        // para todos os símbolos nt
        bison ← bison + +nt + +' :'   //'++' - concatenação de strings
        rhss ← g[nt][rhss]
        for rhs ∈ rhss do                       //para todos os rhs de nt
            bison ← bison + +' |'
            for simb ∈ rhs do     // para todos os símb. T ∪ NT ∈ rhs
                if simb ∈ g.nt then                        //se é símbolo NT
                    bison ← bison + +simb
                if simb ∉ g.nt and is_single_char(simb) then
                    bison ← bison + +'simb'
                    addToSpecials(simb)  //juntar aos caract. especiais
                if ... is_multi_char(simb) then
                    bison ← bison + +simb
                    addToTokens(simb)
        bison ← bison + +';'
    reestruturar(bison)
    EscreverBison(bison, tokens, F)
    EscreverFlex(flex, tokens, specials, L)
    Compila(F, L)
```

Com base nas gramáticas parciais, conjunto de símbolos terminais, caracteres especiais, as funções **EscreverBison** e **EscreverFlex** criam os ficheiros bison e flex, começando por escrever os respectivos cabeçalhos genéricos, garantindo que a sintaxe concreta, proteções de símbolos especiais, tratamento de erros, são corretamente verificados, de modo a que não haja erros na ferramenta gerada.

A função **validsent** avalia se uma determinada frase f pertence à linguagem gerada por uma determinada gramática g. Para tal, a frase a ser analisada é submetida à ferramenta criada por **bisonflex(g)** sendo a resposta final determinada em função do '*ok*' ou '*not ok*' que a ferramenta escrever.

5.3 Geração de frases

Como foi anteriormente referido, a nossa estratégia geral de geração de exercícios passa por criar vários elementos ligados a linguagens e usá-los para preenchimento de enunciados, exemplos, resoluções, sistema de validações. Neste contexto, a geração de frases é uma peça crítica na atividade de geração de exercícios.

Na realidade a geração de frases a partir de uma gramática, não é tarefa fácil. Dada uma gramática, podemos querer frases com características diferentes (há várias funções de geração pertinentes):

- dada uma gramática gerar uma frase qualquer;
- dada uma gramática gerar n frases diferentes;
- dada uma gramática gerar a frase mais curta;
- dada uma gramática gerar todas as frases com comprimento menor que k;
- dada uma gramática gerar todas as frases cuja árvore tem profundidade menor que k;
- dada uma gramática gerar frases *plausíveis* – i.e. que têm as características de dimensão e composição esperadas de acordo com o enredo concreto.

O algoritmo base de um gerador de frases a partir de uma gramática, consiste em partir do axioma, e expandir sucessivamente os símbolos não terminais até obter uma frase final (seguir uma cadeia de derivação). Cada vez que um símbolo não terminal tiver mais que uma produção associada, será sorteada uma delas.

Não discutiremos por agora a geração de símbolos terminais com valor associado (Ex: INT para designar qualquer número inteiro, mas que em situação de geração poderemos pretender que produza um inteiro concreto aleatório).

O algoritmo seguinte descreve o processo natural de geração (embora ignorando alguns potenciais problemas de convergência que só mais tarde discutiremos).

Algorithm 6: sentencegen(g,a): naive TopDownSentGeneration

```
Input: g: gramática
Input: a: ∈ NT ∪ T (default=axiom)
Output: r: frase
begin
    if a ∈ T then                          //geração de simbolo terminal
        return a
    if a ∈ NT then
        r ←' '
        rhss ← g[a]
        rhs ← choice(rhss)                 //seleção aleatória de produção
        for s ∈ rhs do  //expande e concatena todos os símbolos do rhs
            aux ← sentencegen(g, s)
            r ← r + +aux
        return r
```

Este algoritmo é perfeitamente utilizável mas como se irá discutir abaixo, pode ter problemas de terminação. Frequentemente precisamos ainda de ter um controlo mais fino sobre as frases geradas.

Uma primeira opção tomada foi permitir acrescentar probabilidade ou pesos às produções da gramática de modo a que se possa indicar o volume de repetições pretendidas, a importância de cada caminho a seguir.

Exemplo

Pretendemos gerar listas de 'a' com comprimento médio de 6. A gramática

$$\begin{array}{rcl} S & \rightarrow & aS \\ & | & a \\ & ; & \end{array}$$

usando o algoritmo 6 em que a escolha do caminho a seguir é aleatória pura (probabilidade 1/2 para cada produção), vai gerar frases cujo comprimento

médio é c_S, a solução da equação

$$c_S = \frac{c_a + c_S}{2} + \frac{c_a}{2}$$

onde c_a é o comprimento de a, ou seja 1. Obviamente

$$c_S = 2$$

Para que o comprimento das frases aumente, precisamos de escolher o caminho a seguir de forma *tendenciosa*. No entanto não há nada nas gramáticas que permita indicar as *tendências* de geração pretendidas.

Se a probabilidade de escolha do ramo recursivo for 5 vezes maior que a do outro, teremos a seguinte gramática pesada:

$$\begin{array}{rcl} S & \rightarrow & \{5/6\}\ aS \\ & | & \{1/6\}\ a \\ & ; & \end{array}$$

O comprimento médio pode ser determinado resolvendo a equação

$$c_S = \frac{5 \times (c_a + c_S)}{6} + \frac{c_a}{6}$$

o que dá

$$c_S = 6$$

ou seja, teremos em média um comprimento de frase 6 – portanto frases mais compridas que as geradas na situação anterior.

Considere-se a seguinte situação: para um exercício pretendemos criar algumas frases exemplo contendo tantos a como b. Vamos usar a gramática já apresentada anteriormente:

$$\begin{array}{rcl} S & \rightarrow & aSbS \\ & | & bSaS \\ & | & \varepsilon \\ & ; & \end{array}$$

Ao correr o algoritmo 6 facilmente se vê que ao gerar S, na maioria dos casos (66%) expandimos para lados direitos com dois S, ou seja o número de S a

expandir vai sendo sucessivamente maior. Trata-se de uma situação de não terminação.

Se determinarmos o comprimento médio de S usando a técnica da equação, obtemos:

$$c_S = \frac{c_a + c_S + c_b + c_S}{3} + \frac{c_b + c_S + c_a + c_S}{3} + \frac{0}{3}$$

o que daria um comprimento negativo

$$c_S = -4$$

Quando a solução da equação de comprimento dá um valor negativo, o algoritmo de geração de frase tem problemas de terminação.

Reduzindo a probabilidade das duas primeiros produções, conseguimos facilmente garantir a terminação e controlar o tamanho médio das frases. Se por exemplo pretendermos comprimento médio $c_S = 6$, a gramática

$$\begin{array}{rcll} S & \rightarrow & \{p\} & aSbS \\ & | & \{p\} & bSaS \\ & | & \{1-2p\} & \varepsilon \\ & ; & & \end{array}$$

dá origem à equação

$$c_S = p \times (c_a + c_S + c_b + c_S) + p \times (c_b + c_S + c_a + c_S) + (1 - 2p) \times 0$$

o que dá $p = \frac{3}{14}$.

Quando lidamos com gramáticas de maior dimensão, constata-se que a generalidade dos símbolos não terminais não necessita de indicação de probabilidades (ou seja a escolha equiprovável das produções tem comportamento correto). No entanto é frequente haver utilidade em controlar o comportamento de geração de alguns símbolos específicos (por questões de tamanho e terminação).

Quando temos uma gramática com vários não terminais, estamos perante uma situação em que cada não terminal tem a sua equação de comprimento associada às suas probabilidades. A gramática gera um sistema de equações que permite determinar os comprimentos médios das suas frases e dos constituintes de frases.

Consideremos a seguinte gramática referente a expressões aritméticas simplificadas:

$$\begin{array}{rcl} E & \rightarrow & \{1-2p\}\ \ Es \\ & | & \{p\}\ \ E+Es \\ & | & \{p\}\ \ E-Es \\ & ; & \\ Es & \rightarrow & \{q\}\ \ (E) \\ & | & \{1-q\}\ \ int \\ & ; & \end{array}$$

O sistema de equações de comprimento correspondente é

$$\begin{aligned} c_E &= (1-2p) \times c_{Es} + 2p \times (c_E + 1 + c_{Es}) \\ c_{Es} &= q \times 1 + (1-q) \times (2 + c_E) \end{aligned}$$

Notemos que este sistema pode ser usado para determinar as probabilidades p e q em função dos comprimentos c_E e c_{Es}.

Seguidamente apresenta-se um exemplo da sintaxe usada para gramáticas pesadas:

```
E  :  0.50!     Es
   |  0.25!     E + Es
   |  0.25!     E - Es

Es :  0.10!     ( E )
   |  0.90!     int
```

Para comodidade de escrita, quando não são definidas probabilidades de produções o sistema assume equi-probabilidades (ou seja todas as gramáticas anteriores podem ser vistas como um caso particular de gramática pesada).

A algoritmo de geração de frases, durante o processo de escolha da produção a ser seguida, vai agora ter em conta as probabilidades especificadas. Seguidamente apresenta-se a função de escolha probabilística que dado um conjunto de elementos (as produções) e as probabilidades associadas, escolhe

um elemento.

Algorithm 7: choicep(es,ps): escolha probabilística

```
Input: es: lista de elementos
Input: ps: probabilidades associadas
Output: e: elemento
begin
    i ← 0
    t ← 0
    v ← rand(1)
    while v < t do
        i ← i + 1
        t ← t + ps[i]
    return es[i]
```

Com base na escolha probabilística, podemos refinar o algoritmo 6, dando origem a uma versão (algoritmo 8) mais robusta e controlável.

Algorithm 8: sentencegen(g,ps,a,l): TopDownSentGeneration

```
Input: g: gramática
Input: ps: probabilidades
Input: a: ∈ NT ∪ T (default=axioma)
Input: l: nível de recursividade (default=0)
Output: r: frase
begin
    if l > maxim then
        ...reiniciar o processo de geração
    if a ∈ T then                          //geração de simbolo terminal
        return a
    if a ∈ NT then
        r ←' '
        rhss ← g[a]
        p ← ps[a]
        rhs ← choicep(rhss, p)             //seleção de produção
        for s ∈ rhs do  //expande e concatena todos os símbolos do rhs
            aux ← sentencegen(g, ps, s, l + 1)
            r ← r + +aux
        return r
```

Nesta versão temos dois novos parâmetros:

- o vector com as probabilidades das produções `ps` para permitir escolha probabilística;

- o nível de recursividade `l` para controlar situações de não terminação;

A utilização do parâmetro vector de probabilidades permite ainda a criação de uma variante do algoritmo que controla a terminação à custa de alterar dinamicamente este vector: quando escolhemos uma produção p reduzimos a probabilidade de a escolher nas invocações recursivas seguintes [14].

5.4 Comparação de gramáticas

Os princípios gerais de comparação de gramáticas independentes de contexto foram já discutidos no capítulo 4.

A ideia principal consiste em:

1. para cada gramática constrói-se um sistema de equações matriciais não lineares onde:

   (a) os terminais são substituídos por matrizes quadradas aleatórias,

   (b) os não terminais são variáveis matriciais;

2. o sistema é seguidamente resolvido numericamente;

3. procede-se à comparação das matrizes correspondentes aos axiomas;

4. repete-se os pontos anteriores até a obtenção de resultados conclusivos.

Para a implementação deste método foram criadas várias funções que seguidamente se enumeram:

- **TMatrixGen** – geração de matrizes aleatórias

- **solverGram** – resolução do sistema de equações matriciais

- **matrixCompare** – comparação de matrizes

- **set...** – para controlar parâmetros diversos

- **gramequiv** – compara duas gramáticas usando as funções anteriores.

5.4.1 Função TMatrixGen

Esta função gera matrizes $\mathcal{N} \times \mathcal{N}$ aleatórias para substituir nos símbolos terminais. Todos os valores da matriz gerada vão estar contidos no intervalo $[0, \eta]$, sendo a constante η determinada de modo a garantir a convergência do método, de acordo com discutido no capítulo 4.

As linguagens de programação oferecem diferentes representações para os números reais, diferindo no espaço ocupado, na precisão e gama de valores representáveis. A escolha de representações com a maior precisão disponível, é fundamental para qualidade da comparação.

O valor $\mathcal{N}$ é normalmente 2 ou 3.

5.4.2 Função solverGram

A função **solverGram** faz a resolução do sistema de equações matriciais anteriormente introduzido no capítulo 4, utilizando o método iterativo.

Na realidade não é feita a construção explícita do sistema de equações, sendo o valor do segundo membro do sistema calculado diretamente a partir da gramática, conforme a algoritmo 9.

Na implementação as operações matriciais estão a usar o módulo Perl Math::MatrixReal (nomeadamente as funções ligadas a soma, multiplicação,

matriz identidade, e outras).

Algorithm 9: solverGram(g,M_T)

```
Input: g: gramática
Input: M_T: matrizes associadas aos símbolos T
Output: matriz
begin
    M ← M_T            //M : matrizes associadas a T ∪ NT
    for nt ∈ g.nt do
        M[nt] ← 0      //valor inicial das matrizes dos nt é 0
    repeat
        for nt ∈ g.nt do
            s ← 0                          //somatório
            for rhs ∈ g[nt].rhss do
                p ← I
                for y ∈ rhs do
                    p ← p × M[y]
                s ← s + p
            M[nt] ← s
    until até que estabilize a matriz associada ao axioma
    return M[g.axioma]
```

O algoritmo real é mais complicado que o apresentado e tem de lidar com situações de exceção, tal como a divergência do método – o algoritmo é interrompido quando a matriz associada ao axioma ultrapassa certos limites sendo nesta situação retornado um valor matricial infinito.

5.4.3 Função matrixCompare

A função de comparação de matrizes é fundamental para critério de terminação do algoritmo de resolução do sistema de equações matriciais (**solverGram**) e para determinação de equivalência de gramáticas.

A comparação de matrizes (*mcomp*) é habitualmente feita através do cálculo de uma norma da diferença de matrizes:

$$mcomp(m_1, m_2) = norma(m_1 - m_2) < \epsilon$$

sendo possível definir várias normas, como por exemplo

$$\|A\|_1 = \max_j \sum_i |a_{ij}|,$$

$$\|A\|_\infty = \max_i \sum_j |a_{ij}|,$$

$$\|A\|_F = \left(\sum_i \sum_j |a_{ij}|^2 \right)^{\frac{1}{2}}.$$

No nosso caso sentimos necessidade de definir um outro critério mais sensível à representação interna dos números reais usada pelo computador. Como é sabido a representação em computadores da generalidade dos números reais é afetada por um erro de arredondamento. Um número real é normalmente representado por um par (mantissa, expoente) tendo tanto a mantissa como o expoente uma representação de tamanho fixo. A título de exemplo a representação de um real em dupla precisão – "double"(correspondente ao usado na nossa implementação) contempla habitualmente: (1 bit para sinal, 11 bits para expoente, 52 bits para mantissa. Neste tipo de representação, a mantissa corresponde a 14 / 15 casas decimais. A mantissa é portanto responsável pela precisão. O expoente pode variar entre cerca de -308 e 308[1] – sendo responsável pela gama de representação. Daqui resultam os seguintes possíveis problemas:

1. Dois números diferentes mas que sejam iguais nas primeiras 15 casas, são indistinguíveis.

2. Do anterior decorre que $a \pm b = a$ se $a > b \times 10^{15}$.

3. Os números que tenham expoente maior que 308 são indistinguíveis de ∞.

4. Os números que tenham expoente menor que -308 são indistinguíveis de 0.

5. Os números que resultem de aritmética envolvendo o anterior podem não ser definidos (o que por vezes em linguagens de programação aparece como *nan* – not a number).

6. A ordem pela qual são feitas as operações aritméticas pode influenciar o resultado.

Havendo imprecisão na representação dos números, o resultado da comparação poderá ser errado – terá que ser usada uma abordagem mais cautelosa do que o simples cálculo da norma da diferença das matrizes.

[1]Expoente com sinal + 10 bits: $\pm 2^{1024}$ ou seja aproximadamente $\pm 10^{308}$

Consideremos a comparação das seguintes matrizes:

$$\begin{pmatrix} 1 & A \\ 1 \times 10^{-30} & B \end{pmatrix} \quad \text{e} \quad \begin{pmatrix} 1 + 10^{-13} & A \\ 1.3 \times 10^{-30} & B \end{pmatrix},$$

sendo $\epsilon = 10^{-13}$. Qualquer norma habitual da diferença destas matrizes é menor ou igual a 10^{-13}. No entanto a diferença entre 1×10^{-30} e 1.3×10^{-30} é muito significativa (30%), e indicativa que as gramáticas correspondentes não são equivalentes. Este problema pode ser contornado usando uma nova medida de diferença entre as matrizes. Para o caso das matrizes 2×2 esta medida calcula-se do seguinte modo:

Algorithm 10: matrixCompare(A,B):

Input: A, B: matriz
Output: conc : bool
begin
  if *A ou B contêm 'nan' ou 'inf'* **then** //∞ e "not a number"
    return 'Erro'
  //Somatório do erro relativo para cada termo das matrizes
  $d = |A11 - B11|/(|A11| + |B11|)$
    $+|A12 - B12|/(|A12| + |B12|)$
    $+|A21 - B21|/(|A21| + |B21|)$
    $+|A22 - B22|/(|A22| + |B22|)$
  if $d < \epsilon$ **then** //Sendo ϵ o erro admissível
    $conc \leftarrow 1$
  else
    $conc \leftarrow 0$
  return *conc*

Naturalmente, para matrizes 3×3 ou maiores, o algoritmo é análogo.

5.4.4 Função gramequiv

A função **gramequiv** vai basear-se numa função auxiliar **simplegramequiv**, que será invocada as vezes necessárias para comparar duas gramáticas de

acordo com os parâmetros de confiança pretendidos.

Algorithm 11: simplegramequiv(g1,g2,P)

Input: g1,g2: gramática
Input: P: parâmetros de configuração
Output: resultado da comparação
begin
  for $t \in g1.T$ **do**
    $M\_T[t] \leftarrow TMatrixGen(t, P.eta)$
  $s1 \leftarrow solverGram(g1, M\_T)$
  $s2 \leftarrow solverGram(g2, M\_T)$
  $v \leftarrow matrixCompare(s1, s2)$ //norma (s1-s2)
  return v

O esquema de **gramequiv** será então

Algorithm 12: gramequiv(g1,g2)

Input: g1,g2: gramática
Output: resultado da comparação
begin
  $P \leftarrow$ ...determinar parâmetros de configuração (g1,g2)
  repeat
    $v \leftarrow simplegramequiv(g1, g2, P)$
  until *situação conclusiva*
  return v

Os parâmetros de configuração referidos podem ser definidos pelo utilizador e são basicamente:

1. Os valores de erro usados na comparação de matrizes.
2. O valor η – máximo valor de componente de matrizes usados na substituição de terminais.
3. Valores ligados ao critérios de terminação – número máximo de iterações,

5.5 *Pretty print* de gramáticas

Embora de importância inferior aos algoritmos anteriormente discutidos, as funções de *Pretty print* de gramáticas, têm utilidade prática no sistema de geração de exercícios.

Como era de esperar o algoritmo consiste em percorrer o conjunto dos símbolos não terminais da gramática e escrever as produções associadas de acordo com o formato e detalhe pretendido.

Naturalmente faz sentido a criação de várias funções de *pretty print*, de modo a cobrir diferentes usos e formatos de saída.

A função principal tem a seguinte estrutura:

Algorithm 13: grampp(g,ax):formata a gramática para impressão;

```
Input: g: gramática
Input: ax : axiom ∈ NT
Output: r: gramática formatada
begin
    ax ← g.axiom                          //Iniciar pelo axioma
    r ← ax + +' →'
    for rhs ∈ g[ax].rhss do
        for s ∈ rhs do
            r ← r + +s
        r ← r + +'NL |'                   //NL - quebra de linha
    ... remove | no final de r
    for (nt ∈ domg) do                    //Para os outros não-terminais
        if nt = ax then
            next
        r ← nt' →'
        for rhs ∈ domg do
            for s ∈ rhs do
                r ← r + +s
            r ← r + +'NL |'
        ... remove | no final de r
    return r
```

Esta função formata a gramática reconhecida para exibir ao estudante desta forma:

```
S → a S
  | a
  ;
```

A produção de saídas para diferentes formatos é, como referido, importante. Para a criação de gramáticas em formato LaTeX optou-se por:

1. criar um estilo LaTeX com um ambiente **gram** e um conjunto de comandos capazes de escrever produções (vazias ou não), eventualmente

com probabilidades, que seja facilmente adaptável às necessidades específicas;

2. escrever uma função **grampptex** que formata a gramática neste estilo de LaTeX.

Capítulo 6

Conclusão

6.1 Conclusão e trabalhos futuros

Nesta dissertação desenvolveu-se e discutiu-se:

- Formalismo de metagramáticas para geração de exercícios;
- Teoremas de distinguibilidade para gramáticas independentes de contexto;
- Algoritmos de comparação de gramáticas independentes de contexto, através de uma transformação de problemas de linguística-matemática em problemas de análise numérica;
- Análise experimental da aplicabilidade de algoritmos de comparação de gramáticas independentes de contexto;
- Um conjunto de ferramentas de processamento de gramáticas.

Como conclusão geral da tese podemos afirmar que as gramáticas independentes de contexto:

- Podem ser incluídas em sistemas de e-learning;
- Ferramentas como as desenvolvidas, permitem automatizar actividades de geração de exercícios com gramáticas e (o que é mais complexo) a sua avaliação.

Na pesquisa futuras, pretendemos abordar as seguintes questões:

1. Utilização do método de Newton para resolução de sistemas de equações matriciais não lineares;

2. Estudo estatístico do número de testes necessário para garantidamente distinguir entre duas gramáticas;

3. Avaliação da probabilidade da geração de pares de gramáticas diferentes não distinguíveis utilizando a substituição por matrizes ($\mathcal{N} \times \mathcal{N}$);

4. Desenvolver verificadores inteligentes que construam informação de feedback ao estudante tão rica quanto possível, indicando quais os erros que foram cometidos (e apontando as partes correctas).

Bibliografia

[1] HypatiaMat – página do projecto. Url:. `http://www.hypatiamat.com/`.

[2] A. V. Aho, R. Sethi, and J. D. Ullman. *Compilers: Principles, Techniques, and Tools*. Addison-Wesley Longman Publishing Co., Inc., Boston, MA, USA, 1986.

[3] J. J. Almeida, I. Araújo, I. Brito, N. Carvalho, G. J. Machado, R. M. S. Pereira, and G. Smirnov. Math exercise generation and smart assessment. In *Workshop of TICAMES (Information and Communication Technology in Higher Education: Learning Mathematics), CISTI-2013*, pages 1014–1019, 2013.

[4] J. J. Almeida, I. Araújo, I. Brito, N. Carvalho, G. J. Machado, R. M. S. Pereira, and G. Smirnov. Passarola: High-order exercise generation. In *CISTI-2013*, pages 763–768, 2013.

[5] J. J. Almeida, E. Grande, and G. Smirnov. Context-Free Grammars: Exercise Generation and Probabilistic Assessment. In *5th Symposium on Languages, Applications and Technologies (SLATE'16)*, volume 51 of *OpenAccess Series in Informatics (OASIcs)*, pages 1–8. Schloss Dagstuhl–Leibniz-Zentrum fuer Informatik, 2016.

[6] J. J. Almeida, E. Grande, and G. Smirnov. Exercise generation on language specification. In *Recent Advances in Information Systems and Technologies*, Advances in Intelligent Systems and Computing, vol. 659, pages 277–286, 2017. WorldCIST'17.

[7] J. J. Almeida, E. Grande, and G. Smirnov. On the comparison of context-free grammars. *Arxiv*, abs/1702.05945, 2017.

[8] M. Almeida, N. Moreira, and R. Reis. Testing equivalence of regular languages. *Journal of Automata, Languages and Combinatorics*, 15(1/2), 2010.

[9] M. Almeida, N. Moreira, and R. Reis. Finite automata minimization algorithms. In Jiacun Wang, editor, *Handbook of Finite State Based Models and Applications, Discrete Mathematics and Its Applications*, pages pp.145–170. Chapman and Hall/CRC Press, 2012.

[10] N. K. Amruth. Generation of problems, answers, grade and feedback – case study of a fully automates tutor. *Journal on Educational Resources in Computing (JERIC)*, 5(3), 2005.

[11] R. Backhouse. *Syntax of programming languages*. Prentice-Hall International, 1979. Prentice-Hall International series in computer science. Englewood Cliffs, N. J.

[12] D. Baudisch, M. Gesell, and K. Schneider. Online exercise system – a web-based tool for administration and automatic correction of exercises. In J. A. M. Cordeiro, B. Shishkov, A. Verbraeck, and M. Helfert, editors, *Computer Supported Education (CSEDU)*, pages 104–110, 2009.

[13] M. V. Belmonte, E. Guzmán, L. Mandow, E. Millán, and J. L. Pérez de la Cruz. Automatic generation of problems in web-based tutors. In L. C. Jain, R. J. Howlett, N. S. Ichalkaranje, and G. Tonfoni, editors, *Virtual Environments for Teaching & Learning*, World Scientific Publishing, Series on Innovative Intelligence 1, pages 237–281, 2002.

[14] E. Bendersky. Generating random sentences from a context free grammar. 2010. `http://eli.thegreenplace.net/2010/01/28/generating-random-sentences-from-a-context-free-grammar`.

[15] P. Cousot and R. Cousot. Grammar analysis, and parsing by abstract interpretation. In T. Reps, M. Sagiv, and J. Bauer, editors, *Program Analysis and Compilation, Theory and Practice: Essays dedicated to Reinhard Wilhelm*, LNCS 4444, page 178–203. Springer-Verlag, December 2006.

[16] P. Cousot and R. Cousot. Grammar semantics, analysis and parsing by abstract interpretation. *Theor. Comput. Sci.*, 412(44):6135–6192, 2011.

[17] F. DeRemer and T. Pennello. Efficient computation of LALR(1) Look-Ahead Sets. *Transactions on Programming Languages and Systems, ACM*, 4(4):615–649, October 1982.

[18] G. Dettori and D. Persico. *Fostering Self-Regulated Learning through ICT*. Institute for Education Technology-National Research Council, Italy, 2011.

[19] C. Donnelly and R. Stallman. *Bison. The YACC-compatible Parser Generator*, 2015.

[20] A. Eisenbach. phpSyntaxTree - drawing syntax trees made easy. Tool page, 2015. `http://ironcreek.net/phpsyntaxtree/`.

[21] A. Guerman, C. Santos, R. Costa, A. Raposo, A. Mendonça, and C. Lopes. Plataforma semente: Tecnologias de informação para o ensino na ubi. In *Engenharias'07 – Inovação & Desenvolvimento*, pages 21–23, 2007.

[22] A. Gurtovoi, A. Guerman, and C. Santos. Sistema de ensino baseado no computador: Gerador automático dos exercícios. In *Engenharia'2005 – Inovação e Desenvolvimento*, pages 21–23, 2005.

[23] Y. Hasebe. RSyntaxTree - yet another syntax tree generator made with ruby and rmagick. Tool page, 2016. `http://yohasebe.com/rsyntaxtree/`.

[24] R. Heckendorn. A grammar for the C- programming language. Technical Report Version S16, 2016. `http://marvin.cs.uidaho.edu/Teaching/CS445/c-Grammar.pdf`.

[25] Q. T. Jackson. Disambiguation as a quantifiable computational process. Technical Report 3, 2000. `https://www.researchgate.net/publication/2401401_Disambiguation_as_a_Quantifiable_Computational_Process`.

[26] S. C. Johnson. YACC yet another compiler compiler. Technical Report CSTR32, Bell Laboratories, Murray Hill, 1975.

[27] R. Madhavan, M. Mayer, S. Gulwani, and V. Kuncak. Automating grammar comparison. In Jonathan Aldrich and Patrick Eugster, editors, *Proceedings of the 2015 ACM SIGPLAN International Conference on Object-Oriented Programming, Systems, Languages, and Applications, OOPSLA 2015, part of SPLASH 2015, Pittsburgh, PA, USA, October 25-30, 2015*, pages 183–200. ACM, 2015.

[28] R. Madhavan, M. Mayer, S. Gulwani, and V. Kuncak. Towards automating grammar equivalence checking. Technical Report 206921, EPFL, 2015.

[29] J. J. Neto. *Introdução à compilação*. Livros Técnicos e Científicos, Rio de Janeiro, 2016. Campus - Grupo Elsevier.

[30] J. Núñez, R. Cerezo, A. Bernardo, P. Rosário, A. Valle, E. Fernández, and N. Suárez. Implementation of training programs in self-regulated learning strategies in moodle format: results of a experience in higher education. *Psicothema*, 23(2):274–281, Apr 2011.

[31] M. C. Paull and S. H. Unger. Structural equivalence of context-free grammars. *Journal of Computer and System Sciences*, 2(4):427–463, 1968.

[32] V. Paxson. *Flex a fast Scanner Generator*. Free Software Foundation, version 2.6.0 edition, 2016.

[33] R. M. S. Pereira, I. Brito, G. Q. Machado, T. Malheiro, E. Vaz, M. Flores, J. Figueiredo, P. Pereira, and A. Jesus. New e-learning objects for the mathematics courses from engineering degrees: Design and implementation of question banks in maple t. a. using latex. *International Journal of Education and Information Tecnologies*, 4:7–14, 2010.

[34] I. M. Rewitzky. Towards automated testing. In *Fifth CAA Conference*, 2001.

[35] A. Salomaa. *Formal Languages*. Acad. Press, 1973.

[36] A. Salomaa and M. Soittola. *Automata-theoretic aspects of formal power series*. Springer, 1978.

[37] M. P. Schützenberger. On a Theorem of R. Jungen. *Proceedings of the American Mathematical Society*, 13(6):885–890, 1962.

[38] R. W. Sebesta. *Concepts of programming languages*. Pearson, 10th edition, 2009.

[39] R. Sedgewick and K. Wayne. *Algorithms and Data Structures*. Princeton University, 2007.

[40] M. Siegel, S. Derry, J. B. Kim, C. Steinkuehler, J. Street, N. Canty, C. Fassnacht, K. Hewson, C. Hemlo, and R. Spiro. Promoting teacher's flexible use of the learning sciences through case- based problem solving on the www: A theoretical design approach. In B. Fishmann and S. O'Connor-Divelbiss, editors, *Fourth International Conference of the Learning Sciences*, pages 273–279. NJ: Erlbaum, Mahwah, 2000.

[41] M. Tomita, T. Mitamura, H. Musha, and M. Kee. *The Generalized LR Parser/compiler Version 8.1: User's Guide*. Memo (Carnegie Mellon

University. Center for Machine Translation). Center for Machine Translation, Carnegie Mellon University, 1988.

[42] A. P. Tomás and J. P. Leal. A cpl-based tool for computer aided generation and solving of maths exercises. In V. Dhal and P. Wadler, editors, *Fifth International Symposium on Pratical Aspects of Declarative Languages, PADL'2003*, LNCS 2562. Springer-Verlag, 2003.

[43] A. P. Tomás, J. P. Leal, and M. A. Domingues. A web application for mathematics education, in advances in web based learning. In *ICWL 2007*, LNCS 4823, pages 380–391. Springer-Verlag, 2007.

[44] A. P. Tomás, N. Moreira, and N. Pereira. Designing a symbolic solver for arithmetic constraints for computer assisted learning. Technical Report DCC-2005-6, DCC-FC & LIACC, Universidade do Porto, 2005.

[45] A. P. Tomás, N. Moreira, and N. Pereira. Designing a solver for arithmetic constraints to support education in mathematics. In *Artificial Intelligence Applications and Innovations, AIAI*, pages 433–44. Springer and IFIP, IFIP Series, Vol. 204, 2006.

[46] A. P. Tomás and P. Vasconcelos. Generating mathematics exercises by computer. Technical Report DCC-2001-6, DCC-FC & LIACC, Universidade do Porto, 2001.

[47] A. Wiles. Modular elliptic curves and Fermat's Last Theorem. *Anals of Mathematics*, 142, 1995.

[48] A. A. Zinov'ev. Complete (rigorouse) induction and Fermat's Greate Theorem. *Logique & Analyse*, 22, 1979.